(2)

Teaching A Computer Simulation Course With Ada

Final Technical Report
Defense Advanced Research Projects Agency
Grant # MDA972-92-J-1019

Edward L. Lamie
California State University, Stanislaus,
May 1993

Teaching A Computer Simulation Course With Ada

Final Technical Report
Defense Advanced Research Projects Agency
Grant # MDA972-92-J-1019

Edward L. Lamie
California State University, Stanislaus
May 1993

Overview

Overview

Introduction

The purpose of this project was to develop a nucleus of routines, along with accompanying documentation, for use in software engineering courses or courses in discrete-event system simulation. The system is called SIMPACK, after the main package that contains most of the simulation modules.

SIMPACK was developed on an MS DOS computer using Meridian Ada, version 4.1.1. In addition to documentation, a floppy disk is included which contains all the Ada routines used, a compiled version of the system, and a template system to create data files for use by SIMPACK.

Documentation

There are four major components to the documentation. The first part is titled *Course Notes*. This part is oriented toward use of the system in a computer simulation course. SIMPACK can be used as a stand-alone system capable of simulating a variety of models. The course notes contain an introduction to simulation and model building, an introduction to SIMPACK usage, and a discussion of 10 case studies. The case studies range in complexity from a simple single-queue, single-server system to complex systems involving branching, merging, and feedback loops. Transparency masters are also included which contain descriptions of the 10 case studies.

The second part of the documentation is titled *User Manual*. The user manual contains an introduction to SIMPACK, definitions of terms and concepts used, discussion on how to develop models and how to prepare a simulation data file for use by SIMPACK. There is also a section on how to interpret the results produced by SIMPACK.

The third part of the documentation is titled *System Manual*. This manual contains technical information about SIMPACK. The purpose of this manual is to provide enough information so that modifications and enhancements could be made to the system. The manual contains discussion about specific Ada routines as well as data storage, system architecture, output, and simulation termination. There are several appendixes which contain information about the statistics gathered, known bugs, and suggested upgrades.

The fourth part of the documentation are source listings of the Ada routines used in this project. There is a main driver called SIMULATE.ADA, and two packages called SIMPACK.ADA and RANDOM.ADA.

Files on Disk

There is a disk included with this package which contains all the files needed to use or modify SIMPACK. Following is a list of all the files on this disk, as well as a brief comment for each file.

File	Comment
SIMULATE.ADA	main driver
SIMPACK.ADA	package containing most simulation routines
RANDOM.ADA	package containing random number generators
BUILD.BAT	batch file containing compilation sequence
SIMULATE.EXE	compiled, ready-to-use version of SIMPACK
SIM.DAT	sample data file for use by SIMPACK
GO.BAT	batch file for data file preparation
RUN.EXE	file used for data file preparation
SIMSETUP.SKW	file used for data file preparation

Computer Simulation

SIMPACK may be used without modification as a lab component in a computer simulation course. Many introductory features of discrete-event systems simulation may be explored using this system. A compiled version of the software is provided, so the system is usable in its present form, without alteration. No special setup is required, although the system could be recompiled to take advantage of features such as a numeric co-processor. The 10 case studies in the *Course Notes* provide a variety of examples that illustrate the capability of SIMPACK. These case studies can be modified for other problem assignments. Transparency masters are included at the end of the *Course Notes* that contain all the case studies and all the figures. The *User Manual* provides detailed information on how to use SIMPACK.

Software Engineering

SIMPACK may be used in a maintenance-oriented software engineering course. The system is complete and is operational, but there are many features that could be added. SIMPACK could be used in conjunction with one of the software engineering textbooks currently available. Ada source code is provided, and the *System Manual* contains a description of many internal features of the system that are essential in a maintenance environment. A desirable feature that could be incorporated into SIMPACK is another type of probability distribution, rather than being limited to uniform and exponential distributions. For example, a normal distribution or a user-defined discrete distribution would be useful additions. Providing a more user-friendly approach is certainly another good feature.

Course Notes

Table of Contents

Preface

Computer simulation is a popular topic at many colleges and universities. One or more courses dealing with computer simulation are taught at many institutions. The popularity of this subject is due to its usefulness in a variety of fields including business, computer science, and engineering. Computer simulation is an effective planning and prototyping tool.

These course notes contain an introduction to the principles of model building and computer simulation. Concepts related to discrete-event systems simulation are emphasized and an Ada-based simulation tool called SIMPACK is used. SIMPACK was developed for use on a variety of computers ranging from mainframes to microcomputers. It is easy to learn, and it is easy to use.

SIMPACK is a simulation tool designed specifically for model building and simulation. It is based on the programming language Ada, so it may be modified or extended as needed. It contains many key features essential for computer simulation, such as generation of random numbers, management of events, collection of statistics, and generation of standard output. SIMPACK is available for immediate use. The beginning SIMPACK user is freed from programming details and is able to concentrate on model building and simulation. After gaining some experience, the SIMPACK user may add features in an incremental fashion.

Two manuals are included with this package. The first is a user manual, which is intended to assist the user with the current version of SIMPACK. The second is a system manual which is intended to assist the Ada programmer in understanding the internal details of SIMPACK. This will be useful in modifying or adding features.

The approach used in these course notes is to discuss modeling and simulation concepts in a systematic and complete manner. Topics are introduced in a logical, but gradual sequence. The order of concepts follows a "building block" approach, where fundamental features are presented early and used throughout these course notes. Ten case studies are used to develop sound modeling and simulation skills. Each case study follows a consistent pattern and incorporates model definition, data file contents, sample output, and discussion of the simulation.

Introductory simulation and modeling concepts are discussed in chapters 1 and 2. Basic terminology and model building concepts are emphasized in these chapters. Chapter 3 contains an introduction to SIMPACK and Case Studies one through three are presented. These case studies cover simple systems and introduce concepts that are used later. For example, uniform and exponential probability distributions are discussed in this chapter. Chapter 4 contains case studies four through six. These case studies explore models involving servers in tandem, probabilistic branching, and requestor merging. Chapter 5, the last chapter, contains case studies seven through ten. These case studies investigate systems with multiple branching and merging, feedback loops, and complex structures.

Chapter 1
Systems, Models, and Simulation

Introduction

The principal goal of these course notes is to develop techniques to model systems and to study the behavior of these models with the SIMPACK simulation tool. The types of systems that we will consider in these course notes are specified later in this chapter.

Fundamental concepts and terminology pertaining to modeling and simulation are discussed in this chapter. The section titled "Systems" defines how we use the terms *systems* and *system components* in these course notes. The section titled "Models" describes common types of models, emphasizing the type of models that we will use. The section titled "Simulation" describes the process of model building and how we study the behavior of models with the computer.

There are many types of systems that can be modeled and many kinds of simulation techniques. This chapter defines the types of systems, models, and simulation techniques that we consider in these course notes.

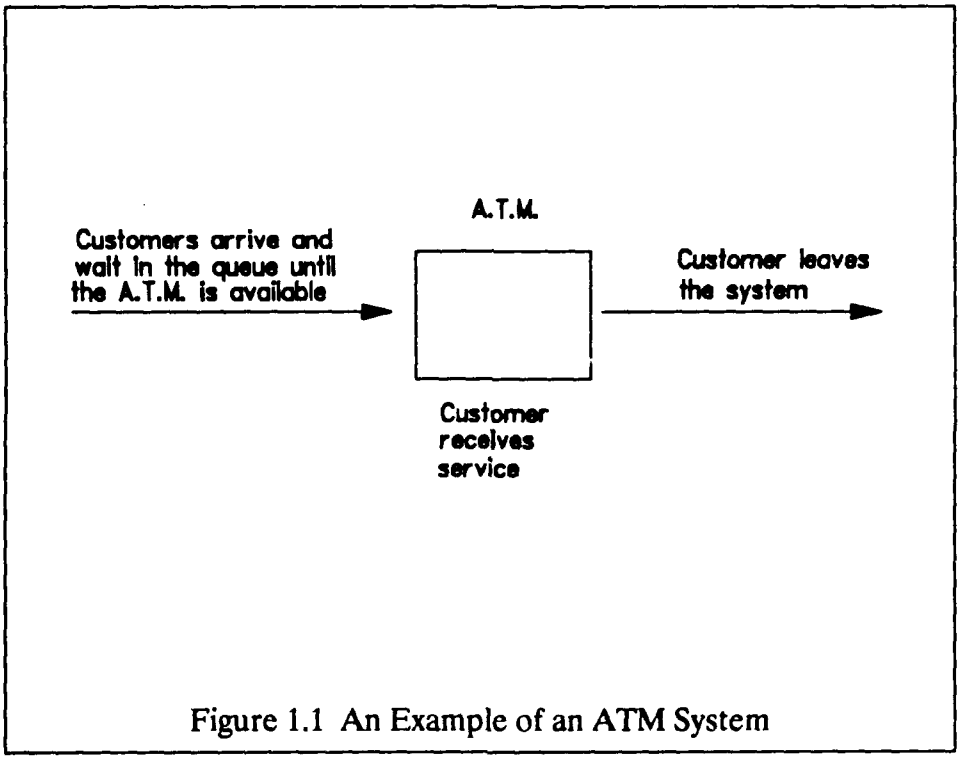


Figure 1.1 An Example of an ATM System

Systems

The term *system* is used in a variety of contexts. We will use the term system to refer to a collection of components that are functionally related, and where changes occur in these functional relationships over time. This is a general definition of the term, but it is sufficient to describe the systems that we wish to model and simulate.

The components of a system are *temporary entities* and *permanent entities*. As the name suggests, a permanent entity exists as long as the system itself does. However, a temporary entity may have a much shorter existence. Each entity may have several associated *attributes* (or values). *Events* (or activities) cause changes to the entities and the corresponding attributes.

An example of such a system is the automatic teller machine (ATM) system in Figure 1.1. In this system, the ATM is represented as a permanent entity and the customers who use the machine are represented as temporary entities. Two events in this system are the arrival of customers and the use of the ATM by a customer. The ATM can accommodate one customer at a time, so when a customer arrives, he or she will join the rear of the *queue* (waiting line). The customer leaves the system after using the ATM.

Models

We are principally interested in developing models of systems that can be simulated on a computer. The activities or events of such systems are usually either deterministic or stochastic in nature. An event is *deterministic* if we are able to predict its behavior exactly when we have the necessary information about that event. An event is *stochastic* if its behavior varies randomly over several possible outcomes. For example, calculating the value of an investment with a fixed interest rate is a deterministic activity. The arrival of customers at a restaurant is an example of a stochastic event because of the apparent random arrival times. We cannot predict with certainty when the next customer will arrive. We will emphasize stochastic systems in these course notes because these systems are used in a wide variety of applications.

Systems are further classified as either *continuous* or *discrete*. Event changes occur gradually (or continuously) over time in a continuous system. Models of continuous systems often involve the use and solution of differential equations. However, event changes occur at specific points in time in a discrete system. The ATM system in Figure 1.1 is an example of a discrete system because all the events occur at specific times. The arrival of a customer does not occur gradually, it happens at a discrete point in time. Similarly, the beginning and completion of ATM use occurs at a specific time.

Consequently, the types of systems that we will model in these course notes are stochastic, discrete event systems. Most of these systems involve queues, which we will model as well. These characteristics are typical of many business and engineering systems.

Computer Simulation

Simulation is the process of creating a model and observing the behavior of that model as it is subjected to various events and activities. A model can take on many forms, ranging from physical representations to computer representations.

Computer simulation is a simulation process, in which the entities of the model are implemented on a computer and the computer is used to imitate the events of the model. A major advantage in using computer simulation is that entities and events can be easily changed. The revised model can be simulated and its behavior analyzed. Consequently, models of present systems can be studied, as well as models of future or proposed systems.

Determining the correctness of a simulation is often a problem. Simulation is not an exact science because there are several simplifications, assumptions, and estimates involved with the process. First, there is the construction of the model itself. The validity of the model is dependent on the skill of the model builder. Sometimes the model builder must make assumptions about the system. For complex systems, the model builder may have to simplify the system in order to represent it. If these steps are not performed correctly, the model is invalid. We must also verify that the computer representation of the model is correct. Even if the model is valid, the model must be correctly implemented on the computer in order for the simulation to be meaningful.

Chapter 2
General Procedures

Introduction

The first step in model building is to identify all the components of the system we are analyzing. We need to determine all the permanent and temporary entities, and their attributes. We also need to recognize all the events of the system and how these events affect the entities.

Drawing a picture or a diagram of the system in question may help clarify the situation. Consider the system depicted in Figure 2.1. This system is essentially the same as the ATM system appearing in Figure 1.1. This system is classified as a *single-queue, single-server system*. There is one entity that gives service to one customer at a time (the single server) and one waiting line for the customers in front of the server (the single queue).

Entities

When given a system, one of the first steps in model building is finding all the permanent entities and all the temporary entities. Consider the system of Figure 2.1. Which entities are part of the system as long as the system remains in

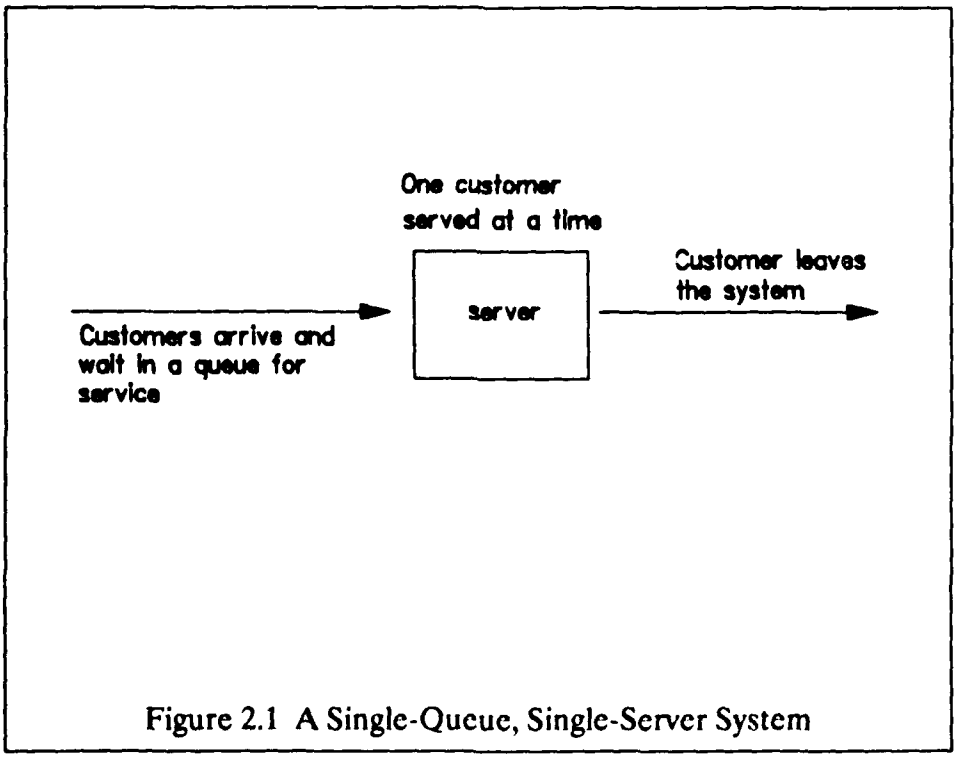


Figure 2.1 A Single-Queue, Single-Server System

existence? There is one permanent entity in this system, the single server. However, there are arbitrarily many temporary entities (the customers) in this system. Customers enter the system and remain until they receive service.

Determining the permanent and temporary entities of a system is usually not a difficult task. Determining the attributes of these entities may not be as straightforward. An attribute is a value or a property of an entity. Attributes add information about entities that is necessary in the simulation process. Because of the simplicity of the system in Figure 2.1, none of the entities have attributes.

Events

Properly identifying the events of a system is the next major step in model building. Events usually involve entities and represent the changes that occur over time in a system. In the system depicted by Figure 2.1, there are two types of events. The events in this system are (1) the arrival of another customer and (2) the service given by the server to the customer. We usually need more information to properly describe these events. In particular, we need to know how often a customer arrives in the system and how long it takes the server to deliver the service. Figure 2.2 is another view of this system, but with this additional information.

According to Figure 2.2, a customer arrives every five to 15 minutes. The arrival time of entities into a system is called the *interarrival time* or *IAT* for short. The ATM serves one customer every four to 10 minutes. The service time of a server in a system is called the *average service time* or *AST* for short. The queue is represented in this figure as a line forming in front of the ATM. We assume that the queue is organized in a *first-in, first-out (FIFO)* fashion. Each customer that arrives joins the rear of the queue and waits for his or her turn with the ATM.

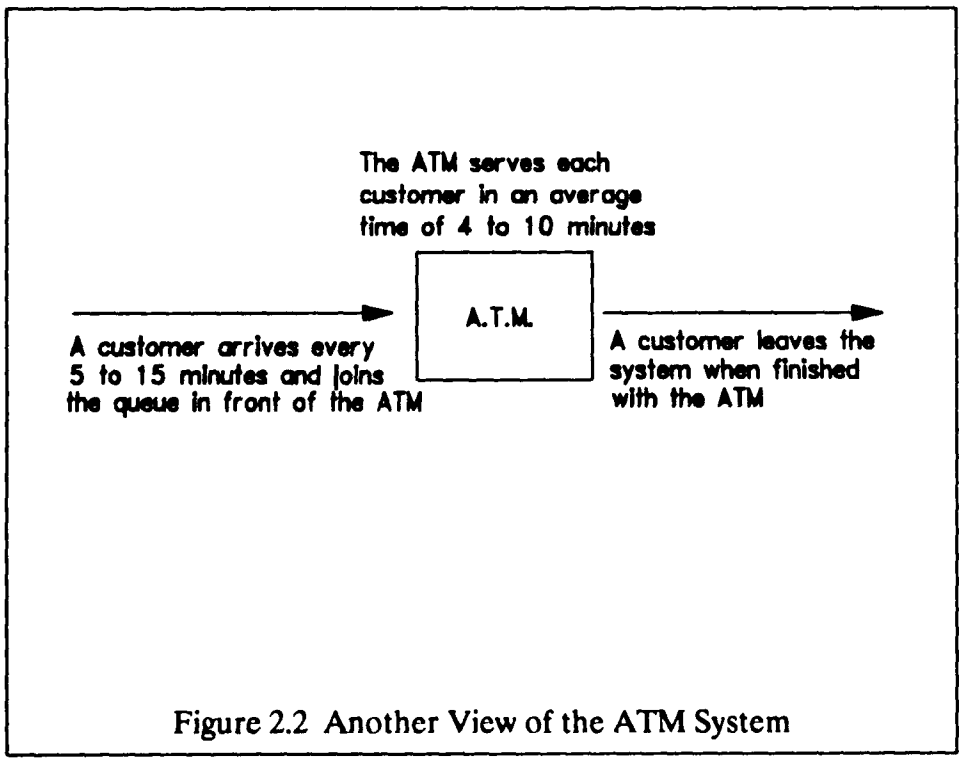


Figure 2.2 Another View of the ATM System

The only information missing in Figure 2.2 is how long the model should be simulated. This could be handled by specifying a certain amount of time to simulate or by indicating the number of customers to be processed. This is an element of information that is required to implement the model on a computer.

Simulation Results

The chief reason for simulating a system is to answer questions or resolve concerns about the behavior of that system. For example, in the ATM system, the following questions could be asked about the queue that forms in front of the ATM.

a. What is the average waiting time in the queue?
b. What was the maximum number of customers in the queue?
c. How many customers spent zero waiting time in the queue?
d. How many customers were in the queue when stimulation stopped?
e. What was the average number of customers in the queue during simulation?

The following questions could be asked about the ATM itself.

a. How many customers were served?
b. What was the average time to serve each customer?
c. How busy was the ATM?

Answers to each of these questions can be obtained as a result of developing a model for this system and simulating the behavior of that model on the computer. In the next chapter, we develop a model for the ATM and show the results of simulating the behavior of this model using SIMPACK.

Chapter 3
Introduction to SIMPACK

Introduction

The User Manual contains the details necessary for installing and running SIMPACK. The basic simulation and modeling concepts used by SIMPACK are discussed in this chapter. We then create a model of the ATM system discussed in the previous chapter, and simulate the behavior of that model using SIMPACK.

Definitions

Following are definitions of several key concepts that we will use when building models with SIMPACK.

Requestor: This is a temporary entity that arrives in the system every so often (arrival time is a random number based on information supplied by you), and waits for service by a server for some amount of time. When the server is available, the requestor spends some amount of time being processed, then either exits the system, or continues on to wait for another server.

Server: This is a permanent entity that requestors wait for. A server holds the requestor for some amount of time to do some sort of processing, then releases the requestor. The requestor can then leave the system, or branch to another server for other processing.

Queue: A queue (or waiting line) forms in front of a server if that server is busy with another requestor.

Time: Time units can represent any time unit that you wish, but you must be consistent in your units when specifying arrival times, service times, lengths of simulations, etc. Units can be seconds, milliseconds, nanoseconds, or even days, weeks, years, or centuries (time units are arbitrary).

I.A.T.: Inter-arrival time: This is the time interval at which requestors arrive in the system. The actual arrival times are random numbers based on the following information:
If you choose a uniform probability distribution, you must supply the minimum and maximum arrival time intervals (for example, a requestor arrives every 3 to 5 minutes).

If you choose an exponential probability distribution, you must supply the mean arrival time (for example, a requestor arrives every 10 seconds, on the average).

A.S.T.: Average service time: This is the length of time that a server holds a requestor for processing. The actual service times are random numbers based on the following information:
If you choose a uniform probability distribution, you must supply the minimum and maximum service times (for example, this server spends between 5 and 15 milliseconds processing each request).
If you choose an exponential probability distribution, you must supply the mean service time (for example, this server spends 62 minutes processing each requestor, on the average).

Capacity: This is the number of requestors that a server can process at the same time.

We will now use these terms in defining the model for the ATM system in Chapter 2.

Case Study # 1 - Building the ATM Model

Figure 3.1 contains a description of a single-queue, single-server system which we use to represent an ATM system. The requestors represent customers with an IAT of eight to 10 time units. This is a uniform distribution which means that the time between customer arrivals is eight, nine, or 10 time units (each of these times is equally likely). The server represents the ATM with an AST of four to six time units. This is also a uniform distribution which means that the time to process each

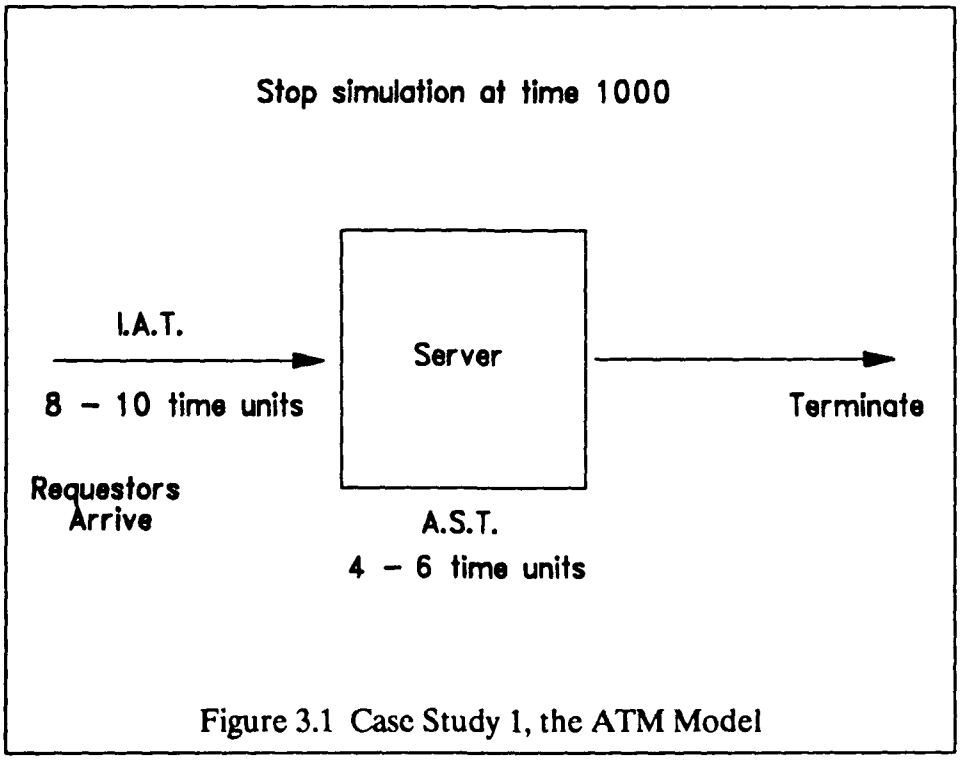


Figure 3.1 Case Study 1, the ATM Model

customer is four, five, or six time units. The simulation will stop when the simulation clock reaches time 1000 (the clock starts at time 0).

Note that we must specify all values as integers when using SIMPACK. The primary reason for this numeric restriction is to improve the execution speed of the simulation process.

Consult the user manual for details of using SIMPACK. Here is a brief overview. First, create a data file called SIM.DAT by using the template that is provided. The purpose of this data file is to configure the model to be simulated. The template would be completed for this problem as follows.

Help: <Alt> H
Quit: <Alt> Q

S I M U L A T I O N S E T U P

I.A.T.	Expon/ Uniform	Mean/ Min	Max
	U	8	10

Stop at time: 1000
Stop # Req.:

Server	Expon / Uniform	A.S.T.: Min	Max	Cap	# servers reached	next server#	prob	next server#
1	U	4	6	1	0			
2								
3								
4								
5								

After entering data in the above template, press <Alt> D to create the data file SIM.DAT. Figure 3.2 contains the contents of that file.

```
t  1000
u  8  10
u 4 6 1 0
```

Figure 3.2
Contents of Data File SIM.DAT

The first line in Figure 3.2 specifies that the simulation will end abruptly at time 1000. The second line specifies the IAT of requestors, i.e., it is a uniform distribution from eight to 10 time units. The third line contains information about server number 1 (the ATM). First it specifies the AST of the server, i.e., it is a uniform distribution from four to six time units. Then the capacity of the server is specified, i.e., one. This means that the ATM can handle only one customer at a time. Finally, the value zero indicates that there are no other servers for the customer to engage. You are not required to use the simulation template to create file SIM.DAT, which is an ASCII text file. However, doing so may be quite error-prone.

```
Simulation Stopped at Time:  1000
1 servers in this run.
SERVER   CAPACITY  ENTRIES  AVG. TIME   MAX    CURRENT   AVG.       UTIL.
                            PER  ENTRY     \     |   /
                                           CONTENTS
 1          1       110        4.9        1      0        0.5         53.9 %


QUEUE    ENTRIES   0-ENTRIES   AVG. TIME      MAX.   CURRENT     AVG.
                                                     CONTENTS
 1         110         110        0.0          1        0        0.0
```

Figure 3.3 Sample Output Produced by SIMPACK

After the SIM.DAT file is prepared, invoke SIMPACK by executing the executable file SIMULATE.EXE which is provided. When simulation is complete, standard output is sent to the default output device, or it may be directed to a file for later use. Figure 3.3 contains a sample of the output produced by SIMPACK for the ATM problem.

The first line of the output in Figure 3.3 indicates that the simulation stopped at time 1000. The next line specifies the number of servers in this model, i.e., one. Information about the server appears next, followed by information about the queue.

The server had 110 entries, that is, 110 customers used the ATM. The average time per customer was 4.9 time units (recall that the AST was four to six time units). There was a maximum of one customer using the ATM at any time (recall that the capacity of the ATM is one). The current contents of the ATM when simulation stopped was zero (no customer was using the ATM at that time). The average contents of the ATM was 0.5 and the utilization of the ATM was 53.9% (this means that the ATM was busy a little over half the time).

The queue had a total of 110 entries and there were 110 zero entries. This means that no customer had to wait for the ATM. The average time spent in the queue by customers was 0.0 and the queue never had more than one customer. The contents of the queue was zero when simulation stopped, and the average contents of the queue was zero (recall that waiting time was zero for all customers).

Exponential Probability Distribution

The uniform probability distribution was used in Case Study 1. The exponential probability distribution is also available with SIMPACK. This provides IAT and AST values that can be close together or far apart. Sometimes this distribution represents real-life distributions better than the uniform distribution. To use an exponential distribution, you specify only a mean value. That value is multiplied by a value returned by the exponential function. Figure 3.4 is a depiction of that function.

Given a mean of eight for either an IAT or an AST, following is a list of possible values that are returned by multiplying eight by various values of the exponential function.

```
 0
 2
 6
11
17
24
36
55
68
```

Despite the large spread in this list, the mean of these values will approach eight if a relatively large sample is used. We will use exponential distributions in the next Case Study.

Case Study # 2 - The ATM Model Revisited

Figure 3.5 contains a description of the ATM model that appeared in the previous Case Study, but with an exponential IAT and AST.

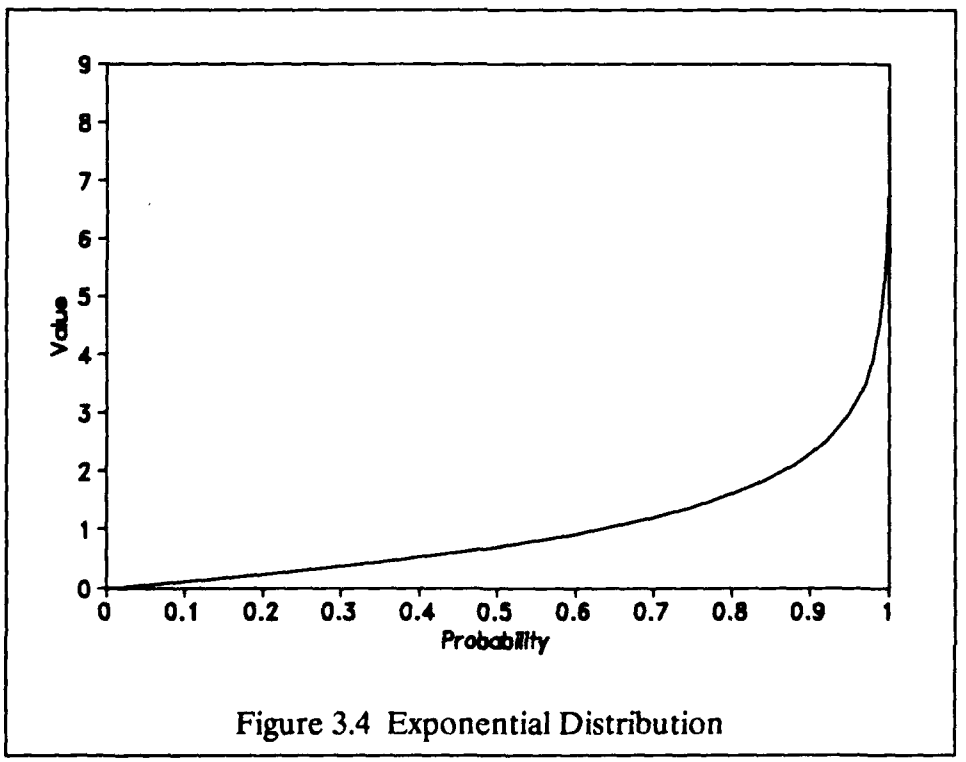


Figure 3.4 Exponential Distribution

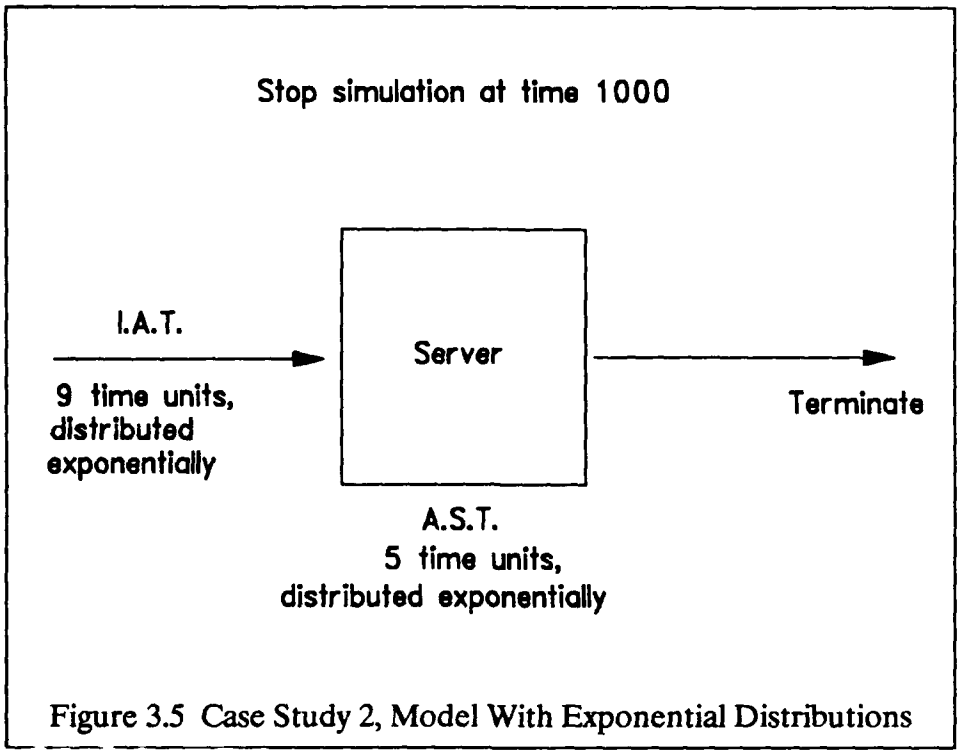


Figure 3.5 Case Study 2, Model With Exponential Distributions

To run this model, we first need to create the data file SIM.DAT which contains a description of the model. Figure 3.6 shows the contents of that data file.

```
t  1000
e  9
e 5  1 0
```

Figure 3.6
Contents of Data File SIM.DAT

The first line of this data file specifies that the simulation is to terminate at time 1000. The second line indicates that the IAT of requestors is distributed exponentially with a mean of nine. The third line specifies that the AST of server 1 i exponentially distributed with a mean of five. The capacity of server 1 is one and there are no servers following it. Figure 3.7 contains the output generated by SIMPACK for this data file.

Note the differences between this output and the Figure 3.3 output. In this simulation, there were only 93 entries each an average time of 5.7 per entry. The sever utilization was 53.0%. Only about half the entries spent no time in queue; the average time spent in the queue was 5.9 time units. There was a maximum of four requestors in the queue during simulation, but no requestor was in the queue when simulation stopped. The average queue contents was 0.6. The variance between

```
Simulation Stopped at Time:  1000
1 servers in this run.
SERVER   CAPACITY  ENTRIES  AVG. TIME   MAX    CURRENT   AVG.       UTIL.
                            PER ENTRY     \     |   /
                                          CONTENTS
 1          1       93        5.7        1      0        0.5         53.0 %


QUEUE   ENTRIES   0-ENTRIES   AVG. TIME      MAX.   CURRENT     AVG.
                                                  CONTENTS
 1         93          48        5.9          4       0        0.6
```

Figure 3.7 Sample Output Produced by SIMPACK

this simulation and the previous simulation with uniform distributions is reasonable. The exponential distribution produces a wide range of IAT and AST values, where the uniform distribution produces a much smaller range of values.

Case Study # 3 - Modeling Server Capacities

The capacity of each server in the previous model was one. However, servers may have capacities much greater than one. We can stop simulation based on the number of requestors that have completed service. In this Case Study, we will model a system with these features. Figure 3.8 contains a diagram of that system.

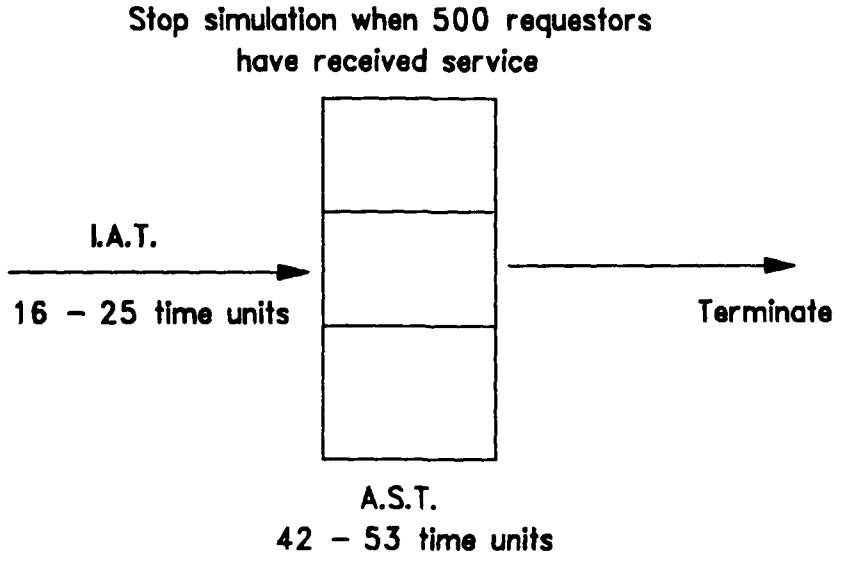


Figure 3.8 Case Study 3, Server With Capacity of 3

The only new features in this model are the server capacity and the method by which simulation is halted. Instead of specifying a capacity of one, we specify a capacity of three. We indicate that simulation is to stop when 500 requestors have received service and left the system (terminated). Figure 3.9 shows the contents of data file SIM.DAT.

```
r  500
u  16  25
u 42 53 3 0
```

Figure 3.9
Contents of File SIM.DAT

The first line in data file SIM.DAT indicates that simulation will halt as a result of a requestor count, rather than at some designated time. The letter "r" (or "R") indicates a requestor count, whereas the letter "t" (or "T") indicates a specific time. The second line indicates that the IAT is a uniform distribution with a low of 16 and a high of 25. The third line shows that the AST is a uniform distribution with a low of 42 and a high of 53. It also indicates that the capacity of server 1 is three, and no other servers follow it. Figure 3.10 contains sample output produced by SIMPACK by using this data file.

```
Simulation Stopped at Time:  10311
1 servers in this run.
SERVER   CAPACITY  ENTRIES  AVG. TIME   MAX    CURRENT   AVG.       UTIL.
                            PER  ENTRY     \     |   /
                                                CONTENTS
 1          3      501        47.4        3      2        2.3         76.8 %

QUEUE    ENTRIES   0-ENTRIES   AVG. TIME      MAX.   CURRENT     AVG.
                                                     CONTENTS
 1         501         500         0.0          1       0        0.0
```

Figure 3.10 Sample Output Produced by SIMPACK

Simulation stopped at time 10311 after the 500th requestor had received service. Each requestor spent an average of 47.4 time units with the server. The server had a utilization of 76.8%. No time was spent in the queue; virtually all the requestors spent zero time waiting.

Chapter 4
Multiple Servers and Branching

Introduction

The examples in Chapter 1 were confined to single-server models. Most models had a capacity of one, although one example illustrated a server with capacity of three. In this chapter, we will consider systems with several servers and probabilistic branching.

Case Study # 4 - Servers in Tandem

In this case study, we consider two servers in tandem. One of the servers has a uniform distribution and the other server has an exponential distribution. Figure 4.1 contains a description of this system.

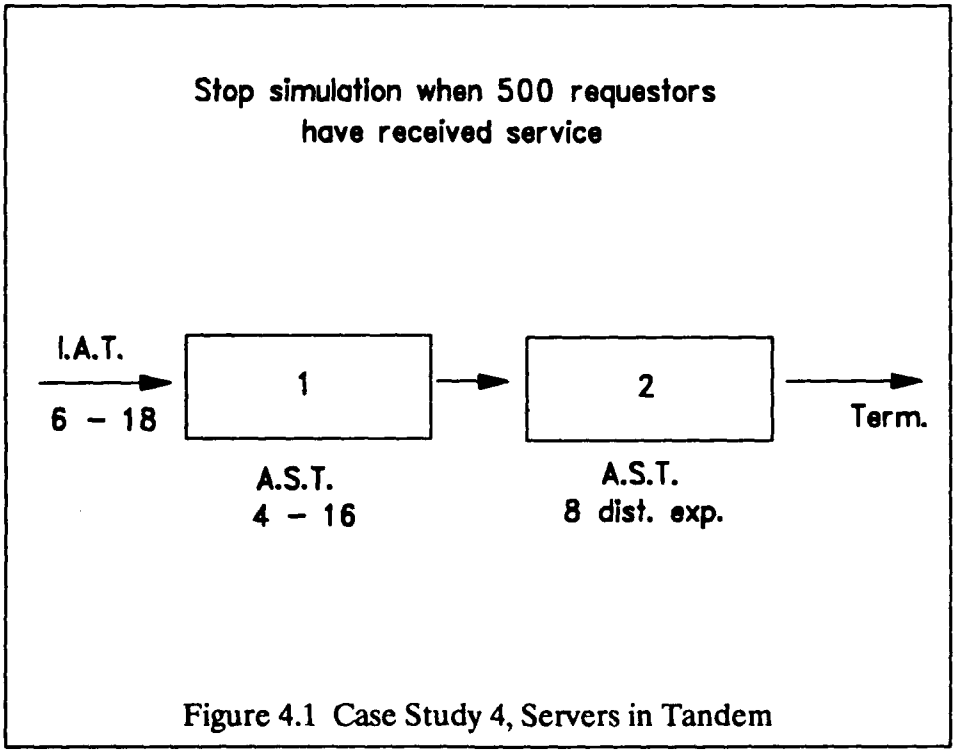


Figure 4.1 Case Study 4, Servers in Tandem

Figure 4.2 contains the contents of data file SIM.DAT.

```
r  500
u  6  18
u 4 16 1 1 2 100
e 8  1 0
```

Figure 4.2
Contents of File SIM.DAT

After running this simulation for 500 time units, the output in Figure 4.3 is produced.

```
Simulation Stopped at Time:  5959
2 servers in this run.
SERVER   CAPACITY  ENTRIES  AVG. TIME   MAX    CURRENT   AVG.      UTIL.
                            PER  ENTRY    \     |    /
                                          CONTENTS
 1          1       501        10.3       1      1        0.9       86.0 %
 2          1       500         8.1       1      0        0.7       67.8 %


QUEUE   ENTRIES   0-ENTRIES   AVG. TIME      MAX.   CURRENT     AVG.
                                                  CONTENTS
 1        501         194        4.8          3       1        0.6
 2        500         265        7.4          6       0        0.7
```

Figure 4.3 Output of Case Study 4

Although the AST of server 2 is less than that of server 1, the maximum contents of queue 2 is six, while the maximum contents of queue 1 is only three. Queue 2 had more zero entries than queue 1, and the average time requestors spent in queue 2 was greater than in queue 1. However, the utilization of server 1 was about 86% while the utilization of server 2 was only about 68%. The explanation for these differences is that server 2 had an AST with an exponential distribution, while server 1 has an AST with a uniform distribution.

Case Study # 5 - Probabilistic Branching

We introduce the concept of probabilistic branching in this case study. When a requestor leaves a server, it can either terminate (leave the system) or branch to

another for additional service. Figure 4.4 contains a depiction of a system with this type of branching.

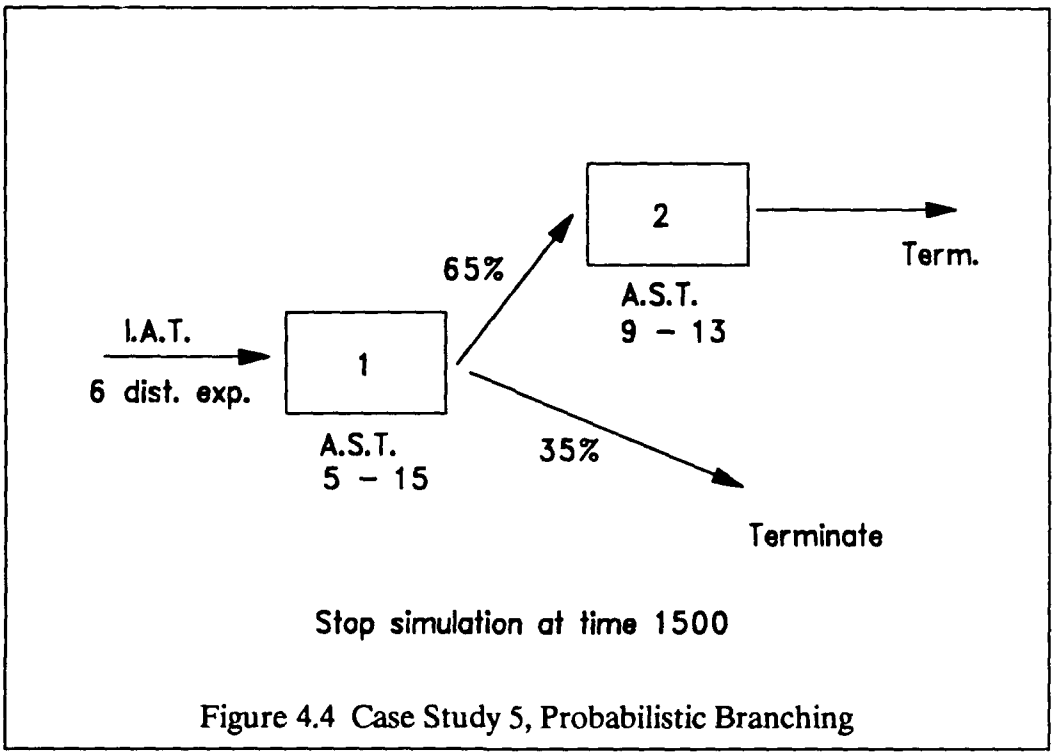


Figure 4.4 Case Study 5, Probabilistic Branching

When a requestor has completed service with server 1, it branches to server 2 with 0.65 probability (although we specify probabilities as percentages). We do not specify a probability that the requestor terminates, because that is the default action. Figure 4.5 contains the data file for this problem.

```
t  1500
e  12
u 5 15 1 1 2 65
u 9 13 1 0
```

Figure 4.5
Contents of File SIM.DAT

After running this simulation for 1500 time units, the output in Figure 4.6 is produced.

```
Simulation Stopped at Time:  1500
2 servers in this run.
SERVER   CAPACITY  ENTRIES  AVG. TIME   MAX    CURRENT    AVG.      UTIL.
                            PER ENTRY    \     |     /
                                                CONTENTS
 1          1        133       10.0       1      1        0.9      89.1 %
 2          1         87       10.8       1      0        0.6      62.7 %


QUEUE   ENTRIES   0-ENTRIES   AVG. TIME     MAX.   CURRENT     AVG.
                                                  CONTENTS
 1         136         11        30.6         9       2        3.5
 2          87         51         2.8         2       0        0.1
```

Figure 4.6 Output of Case Study 5

Server 1 should be able to process requestors faster than they arrive, i.e., the IAT is about 12 and the AST is about 10. However, only 11 requestors spent zero time in queue 1. Furthermore, queue 1 had a maximum of nine requestors waiting, but only two were waiting when simulation stopped. One explanation for this situation is that many requestors were created when simulation started, and the server spent most of the time trying to catch up.

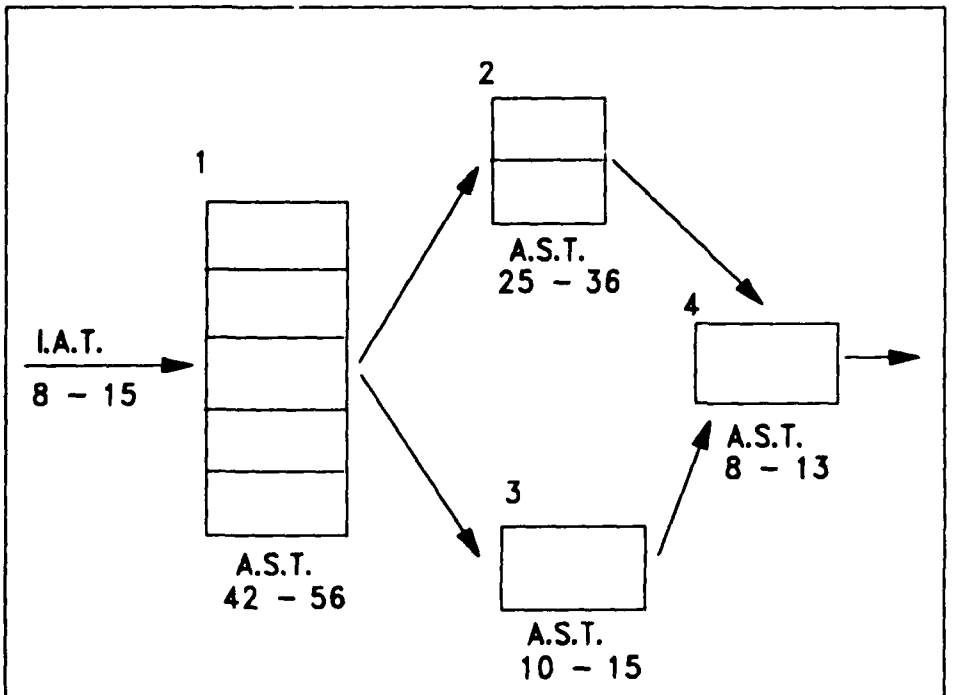


Figure 4.7 Case Study 6, Merging Requestors

Case Study # 6 - Merging Requestors

In this case study, requestors branch to two different servers, then branch to a common server. Figure 4.7 contains a description of this system. When a requestor completes service in server 1, it then branches to either server 2 or server 3. When a requestor finishes service with either server 2 or server 3, it then branches to server 4, then leaves the system. Simulation stops after 200 requestors have received service. Figure 4.8 contains the contents of the data file.

```
r  200
u  8  15
u 42 56 5 2 2 22 3 78
u 25 36 2 1 4 100
u 10 15 1 1 4 100
u 8 13 1 0
```

Figure 4.8
Contents of File SIM.DAT

This simulation stops after 200 requestors have received service. Figure 4.9 contains the output generated by this simulation.

```
Simulation Stopped at Time:  2397
4 servers in this run.
SERVER   CAPACITY  ENTRIES  AVG. TIME   MAX    CURRENT   AVG.      UTIL.
                            PER  ENTRY    \     |    /
                                                CONTENTS
 1          5       203        50.4       5       4        4.3       85.3 %
 2          2        50        30.4       2       1        0.6       31.7 %
 3          1       151        12.7       1       1        0.8       80.1 %
 4          1       201        10.6       1       1        0.9       87.9 %


QUEUE   ENTRIES   0-ENTRIES   AVG. TIME      MAX.   CURRENT     AVG.
                                                   CONTENTS
 1         207        184         0.4          1        0         0.0
 2          51         51         0.0          1        0         0.0
 3         152         56         6.5          3        0         0.4
 4         201         55         5.7          2        1         0.6
```

Figure 4.9 Output of Case Study 6

The utilization of the servers corresponds closely with the average time spent in the respective queues. Figure 4.10 contains a comparison of the server utilizations for this problem.

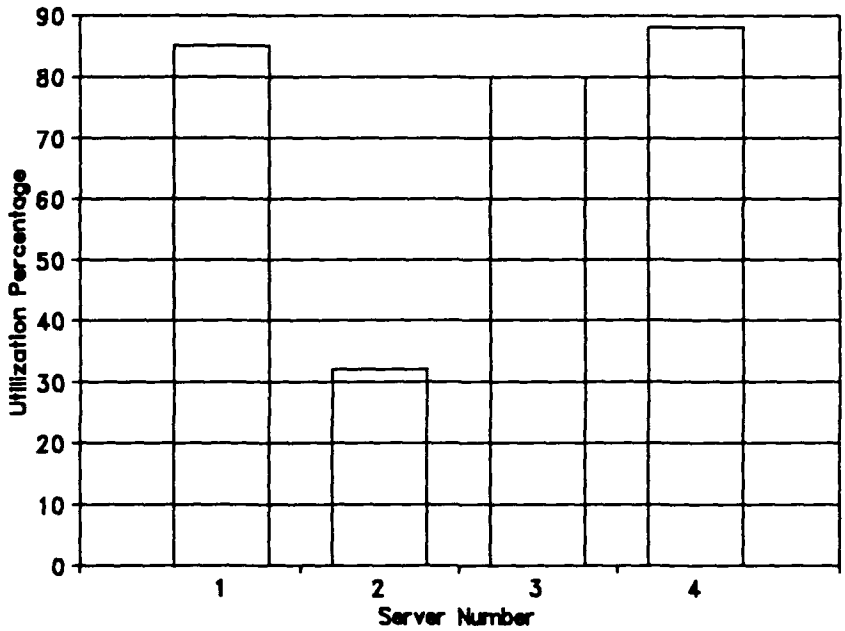


Figure 4.10 Server Utilizations for Case Study 6

Chapter 5
Complex Branching and Feedback Loops

Introduction

In this chapter, we explore models of increasing complexity. Case studies 7 and 8 illustrate 4-way branching and cross-branching. Case studies 9 and 10 investigate two approaches for feedback loops.

Case Study # 7 - Complex Branching

In this case study, a requestor is sent to one of four servers after receiving service at server 1. The capacity of the servers ranges from one to three. When a requestor finishes server 4, it is sent to either server 6 or 7. Similarly, when a requestor finishes service at server 5, it is sent to either server 6 or 7, although with different probabilities. Figure 5.1 contains an illustration of this model.

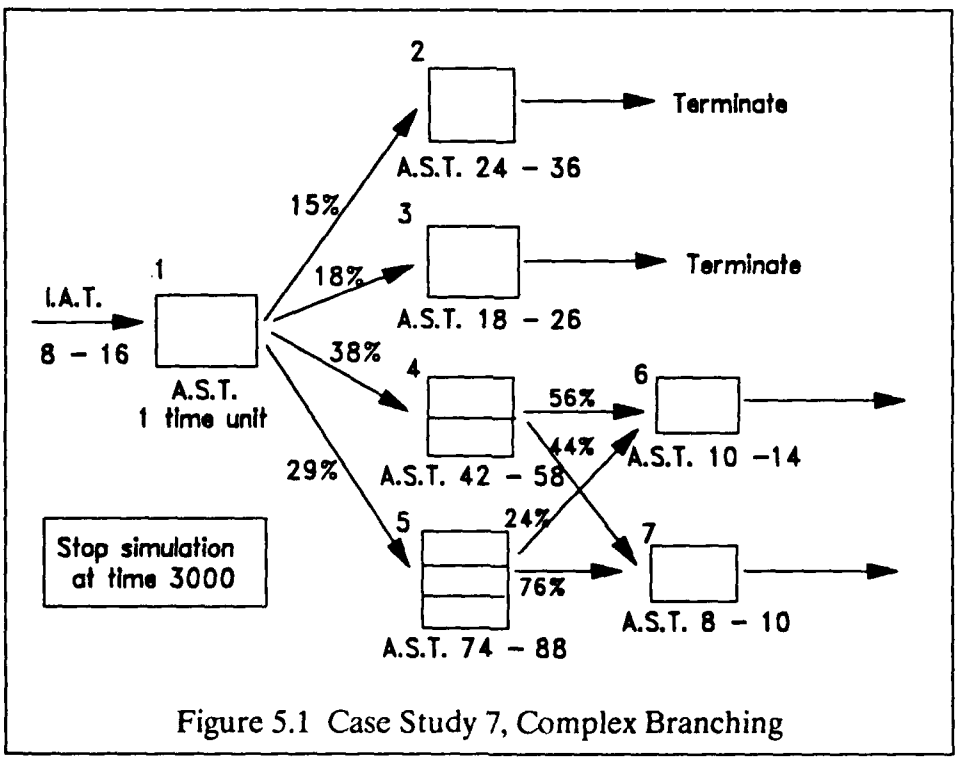


Figure 5.1 Case Study 7, Complex Branching

Figure 5.2 contains the contents of the data file for this case study.

```
t  3000
U  8  16
U 1 1 1 4 2 15 3 18 4 38 5 29
u 24 36 1 0
u 18 26 1 0
u 42 58 2 2 6 56 7 44
u 74 88 3 2 6 24 7 76
u 10 14 1 0
U 8 10 1 0
```

Figure 5.2
Contents of File SIM.DAT

Simulation of this model stops at time 3000. Figure 5.3 contains the output generated by this simulation.

```
Simulation Stopped at Time:  3000
7 servers in this run.
SERVER   CAPACITY  ENTRIES  AVG. TIME   MAX    CURRENT   AVG.      UTIL.
                            PER ENTRY    \     |    /
                                                CONTENTS
 1          1       251        1.0        1       0        0.1       8.4 %
 2          1        34       30.9        1       0        0.4      35.0 %
 3          1        53       22.7        1       0        0.4      40.1 %
 4          2        99       50.8        2       0        1.7      83.8 %
 5          3        62       81.4        3       3        1.7      56.1 %
 6          1        74       12.1        1       0        0.3      29.9 %
 7          1        87        8.9        1       0        0.3      25.9 %


QUEUE   ENTRIES   0-ENTRIES   AVG. TIME      MAX.   CURRENT     AVG.
                                                   CONTENTS
 1        251        251         0.0          1        0        0.0
 2         34         29         1.9          1        0        0.1
 3         53         44         2.1          2        0        0.1
 4         99         38        19.0          4        0        0.6
 5         65         58         1.0          2        0        0.0
 6         74         57         1.9          2        0        0.1
 7         87         70         1.0          1        0        0.1
```

Figure 5.3 Output Produced by Case Study 7

Case Study # 8 - Complex Branching and Merging

Figure 5.4 depicts a model that is considerably more complex than any we have explored thus far. Only the IAT value for the requestors appears in this figure; the AST values for the servers appear separately.

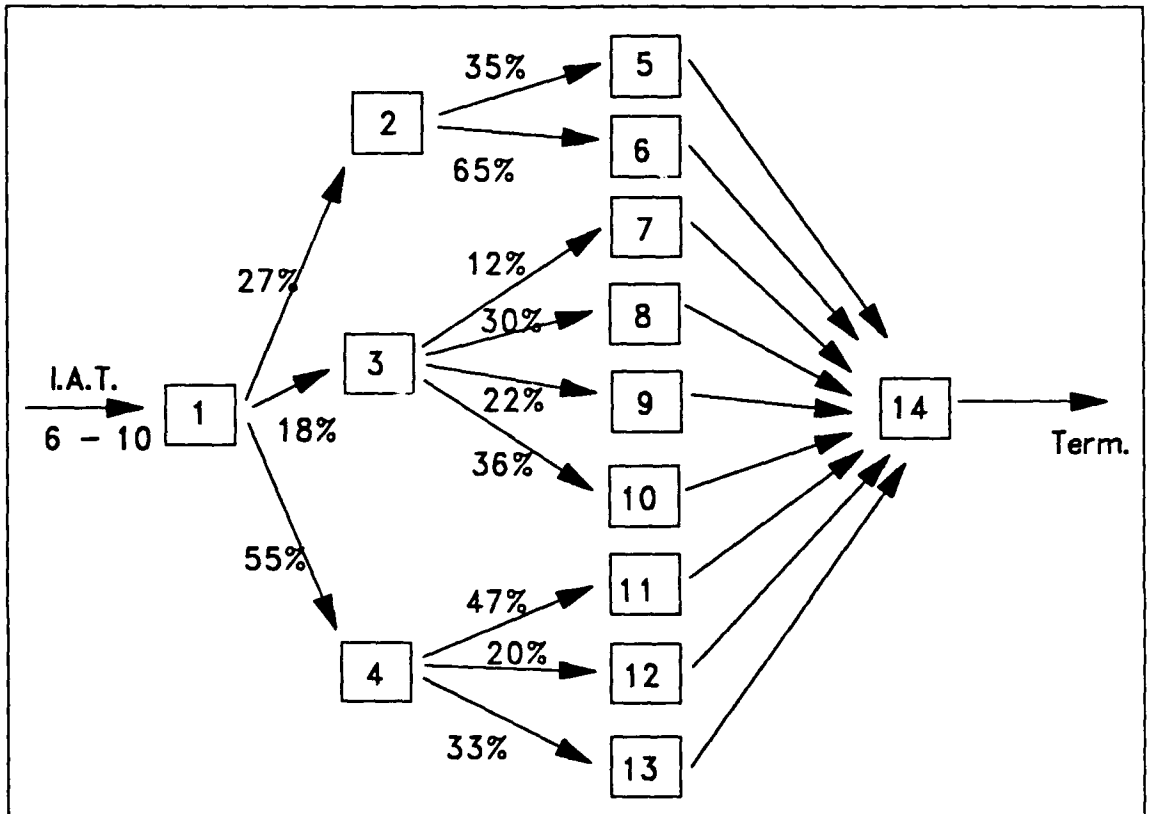


Figure 5.4 Case Study 8, Complex Branching and Merging

Following are the AST values for the servers.

Server Number	A.S.T. (time units)
1	2 - 6
2	6 - 8
3	8 - 12
4	4 - 6
5	12 - 16
6	14 - 18
7	22 - 26
8	16 - 18
9	18 - 20
10	12 - 14
11	6 - 8
12	8 - 10
13	7 - 9
14	1 - 3

Figure 5.5 contains the contents of the data file for this case study.

```
r  200
u  6  10
u 2 6 1 3 2 27 3 18 4 55
u 6 8 1 2 5 35 6 65
u 8 12 1 4 7 12 8 30 9 22 10 36
u 4 6 1 3 11 47 12 20 13 33
u 12 16 1 0
u 14 18 1 0
u 22 26 1 0
u 16 18 1 0
u 18 20 1 0
u 12 14 1 0
u 6 8 1 0
u 8 10 1 0
u 7 9 1 0
u 1 3 1 0
```

Figure 5.5
Contents of File SIM.DAT

Simulation of this model stops when 200 requestors have received service. Figure 5.6 contains the server statistics generated by this simulation.

```
Simulation Stopped at Time:  1608
13 servers in this run.
```

SERVER	CAPACITY	ENTRIES	AVG. TIME PER ENTRY	MAX CONTENTS	CURRENT CONTENTS	AVG. CONTENTS	UTIL.
1	1	201	4.1	1	1	0.5	50.6 %
2	1	44	7.3	1	1	0.2	20.0 %
3	1	42	10.2	1	0	0.3	26.6 %
4	1	113	5.1	1	0	0.4	35.9 %
5	1	16	14.2	1	0	0.1	14.1 %
6	1	28	15.6	1	0	0.3	27.2 %
7	1	2	24.0	1	0	0.0	3.0 %
8	1	16	16.9	1	0	0.2	16.9 %
9	1	9	19.4	1	0	0.1	10.9 %
10	1	15	13.0	1	0	0.1	12.1 %
11	1	49	6.9	1	0	0.2	21.0 %
12	1	22	8.6	1	0	0.1	11.8 %
13	1	42	8.0	1	0	0.2	20.8 %

Figure 5.6 Server Statistics for Case Study 8

Figure 5.6 contains the server statistics generated by this simulation.

QUEUE	ENTRIES	0-ENTRIES	AVG. TIME	MAX.	CURRENT CONTENTS	AVG.
1	201	201	0.0	1	0	0.0
2	45	44	0.0	1	0	0.0
3	42	32	1.1	1	0	0.0
4	113	108	0.1	1	0	0.1
5	16	14	0.2	1	0	0.0
6	28	20	3.6	2	0	0.1
7	2	2	0.0	1	0	0.0
8	16	14	0.4	1	0	0.0
9	9	6	4.7	1	0	0.1
10	15	12	0.6	1	0	0.0
11	49	45	0.2	1	0	0.0
12	22	22	0.0	1	0	0.0
13	42	36	0.4	1	0	0.0

Figure 5.7 Queue Statistics for Case Study 8

Case Study # 9 - Using a Feedback Loop

In this case study, a requestor returns to a server that it has previously engaged. This is a simple case of feedback, as depicted in Figure 5.8.

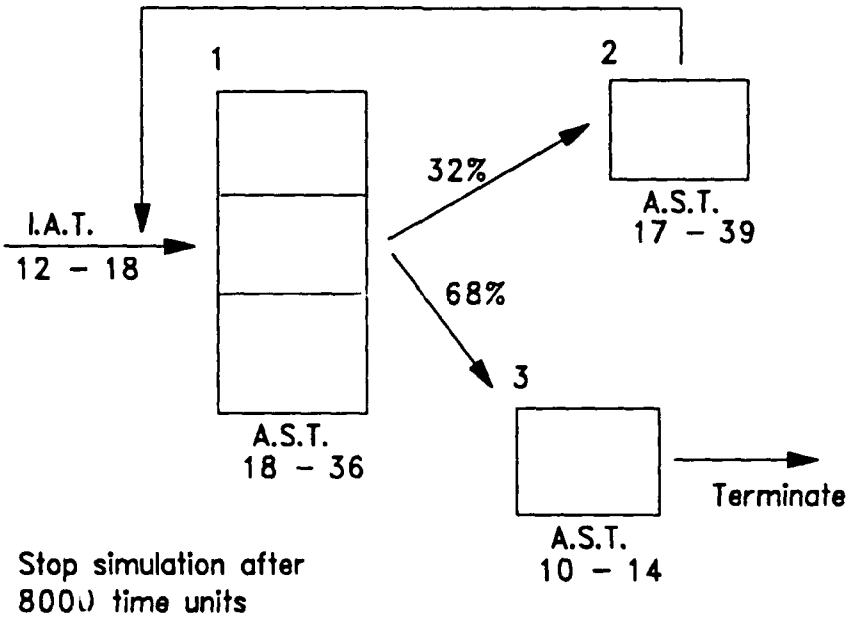


Figure 5.8 Case Study 9, Using a Feedback Loop

This illustrates the ability of a requestor to repeatedly engage any server in the system. In this model, a requestor might engage server 1 arbitrarily many times. Figure 5.9 shows the data file for this model.

```
t  8000
u  12  18
u 18 36 3 2 2 32 3 68
u 17 39 1 1 1 100
u 10 14 1 0
```

Figure 5.9
Contents of SIM.DAT

Simulation of this model stops at time 8000. Figure 5.10 contains the output generated by this simulation.

```
Simulation Stopped at Time:  8000
3 servers in this run.
SERVER   CAPACITY  ENTRIES  AVG. TIME   MAX    CURRENT   AVG.      UTIL.
                            PER  ENTRY    \     |    /
                                          CONTENTS
 1          3       777       27.1       3      3        2.6       87.9 %
 2          1       256       28.1       1      1        0.9       90.0 %
 3          1       520       12.0       1      0        0.8       78.3 %


QUEUE   ENTRIES   0-ENTRIES   AVG. TIME      MAX.   CURRENT     AVG.
                                                  CONTENTS
 1         780         457        2.5          3       0         0.1
 2         257          32       79.9         10       0         2.9
 3         520         184        6.8          4       0         0.4
```

Figure 5.10 Output Produced by Case Study 9

The actual number of requestors to enter this system was probably about 530. Because of the feedback loop in this model, server 1 was engaged 777 times.

Case Study # 10 - Complex Structures

Our last case study involves several complex probabilistic branches. Figure 5.11 illustrates this model.

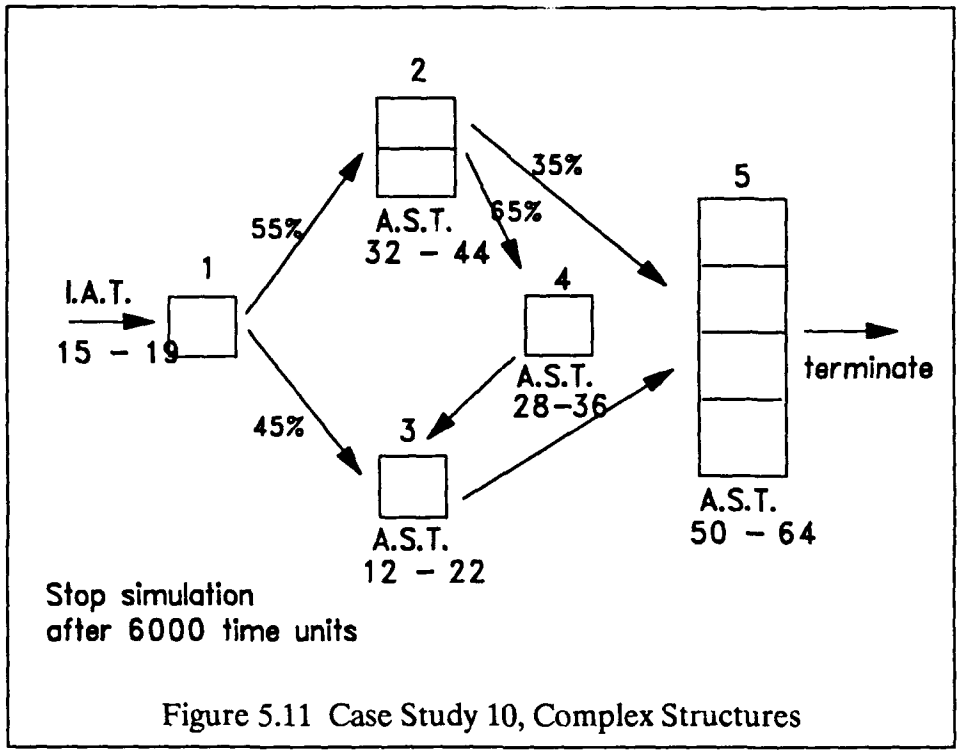


Figure 5.11 Case Study 10, Complex Structures

This illustrates the ability of a requestor to take several different paths through the model. Figure 5.13 shows the data file for this model.

```
t  6000
u  15  19
u 12 18 1 2 2 55 3 45
u 32 44 2 2 4 65 5 35
u 12 22 1 1 5 100
u 28 36 1 1 3 100
u 50 64 4 0
```

Figure 5.12
Contents of Data File

Simulation of this model stops at time 6000. Figure 5.13 contains the output generated by this simulation.

```
Simulation Stopped at Time:  6000
5 servers in this run.
```

SERVER	CAPACITY	ENTRIES	AVG. TIME PER ENTRY	MAX	CURRENT CONTENTS	AVG.	UTIL.
1	1	352	14.9	1	1	0.9	87.4 %
2	2	178	37.8	2	1	1.1	56.1 %
3	1	289	17.3	1	1	0.8	83.2 %
4	1	117	32.0	1	1	0.6	62.5 %
5	4	345	57.4	4	3	3.3	82.5 %

QUEUE	ENTRIES	0-ENTRIES	AVG. TIME	MAX.	CURRENT CONTENTS	AVG.
1	353	271	0.6	1	0	0.1
2	179	137	1.8	2	0	0.1
3	290	65	16.7	5	0	0.9
4	119	64	11.9	3	1	0.2
5	348	258	2.3	3	0	0.2

Figure 5.13 Output Produced by Case Study 10

There is a considerable range of server utilizations produced by simulating this model. Figure 5.14 contains a chart comparing these utilizations.

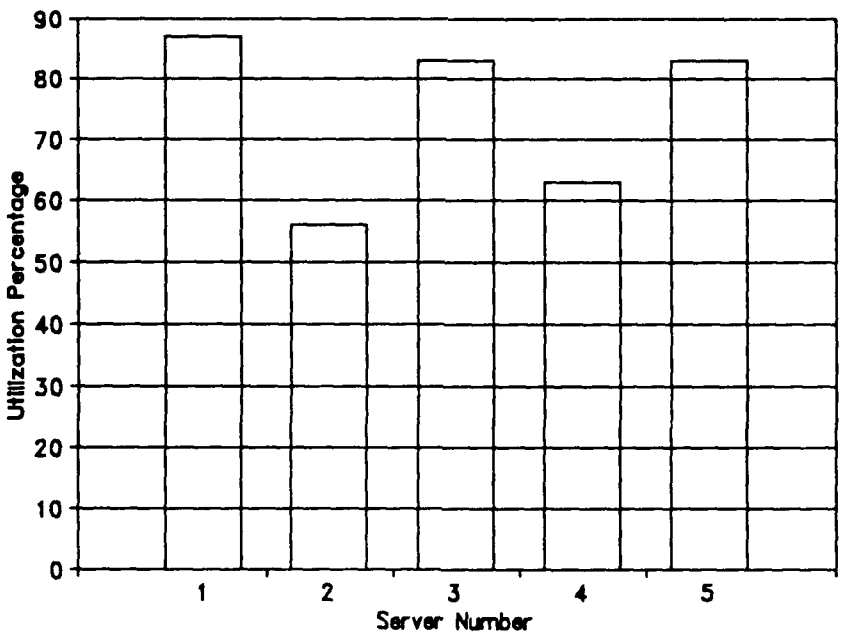


Figure 5.14 Comparison of Server Utilizations in Case Study 10

Appendix A - Bibliography

"Ada experience in the undergraduate curriculum", Communications of the ACM, Nov 1992 v35 n11, p53

"An assessment procedure for simulation models: a case study", Operations Research, Sept-Oct 1991 v39 n5, p710

Banks, Jerry, Discrete-event system simulation, 514 pages, illus., Englewood Cliffs, N.J.,Prentice-Hall, c1984

Chelini, J., "An Example of Event-Driven Asynchronous Scheduling", Ada letters, 07/01/90

Chorafas, Dimitris N., Systems and simulation, 503 pages, illus., New York, Academic Press, 1965

De Acetis, Louis A., "Using ADA tasks to stimulate operating equipment" Computers in physics, 09/01/90

Devlin, Michael J.I., "Ada Technology for Simulation", National defense, 11/01/90

Franta, W. R., The process view of simulation, 244 pages, illus., New York : North-Holland, c1977

Goldsack, Stephen J., Ada for specification : possibilities and limitations, 265 pages, illus., Cambridge, New York, Cambridge University Press, 1985

Gonzalez, D.W., "Multitasking Software Components.", Ada letters, 01/01/90

Hoover, S. and Perry R., Simulation: A Problem-Solving Approach, Addison-Wesley, 1989

McGettrick, Andrew D., Program verification using Ada, 345 pages, Cambridge; New York : Cambridge University Press, 1982

Mize, Joe H., Essentials of simulation, 234 pages, illus., Englewood Cliffs, N.J., Prentice-Hall (1968)

"Modeling reality (The science of computing)", American Scientist, Nov-Dec 1990 v78 n6, p495

Murray, A.G., "Ada Tasking as a Tool for Ecological Modelling.", Ada letters, 09/01/90

"New simulation software", Aviation Week & Space Technology, April 20, 1992 v136 n16, p13

"Parallel Ada in simulation systems.", Defense Electronics, Nov. 1992 v24 n11, p35

"Planning queueing simulations", Management Science, Nov 1989 v35 n11, p1341

"Prediction and prescription in systems modeling", Operations Research, Jan-Feb 1990 v38 n1, p7

Quiggle, Thomas J., "Efficient Periodic Execution of Ada Tasks", Ada letters, Fall 90

"Random numbers for simulation. (Discrete Event Simulation)", Communications of the ACM, Oct 1990 v33 n10, p85

Roos, J. "A Real-Time Support Processor for Ada Tasking", Sigplan notices, 05/01/89

"Secrets of successful simulation studies", Industrial Engineering, May 1990 v22 n5, p47

"Sensitivity analysis and performance extrapolation for computer simulation models", Operations Research, Jan-Feb 1989 v37 n1, p72

"Simulation software gains sophistication: control-system simulation", EDN, April 23, 1992 v37 n9, p79

Sjoland, M., Thyselius R., and Sjoland, B., "Adam an Ada Simulation Library", Tri-Ada 92 Conference Proceedings, pages 108-115

Solomon, Susan L., Simulation of waiting-line systems, 452 pages, illus., Englewood Cliffs, N.J. Prentice-Hall, c1983

Sommerville, Ian, Software Engineering, Fourth Edition, Addison-Wesley, 1992

Teaching a Software Engineering Project Course, Software Engineering Institute, Course Notes, 1992

Tocher, K. D., The art of simulation, 184 pages, illus., London, English Universities Press (1973, 1967)

"Using computer simulation as a model for classroom activity", The Social Studies, Sept-Oct 1987 v78 n5, p225

Weiss, Mark A., Data Structures and Algorithm Analysis in Ada, Benjamin/Cummings, 1993

Appendix B

Transparency Masters

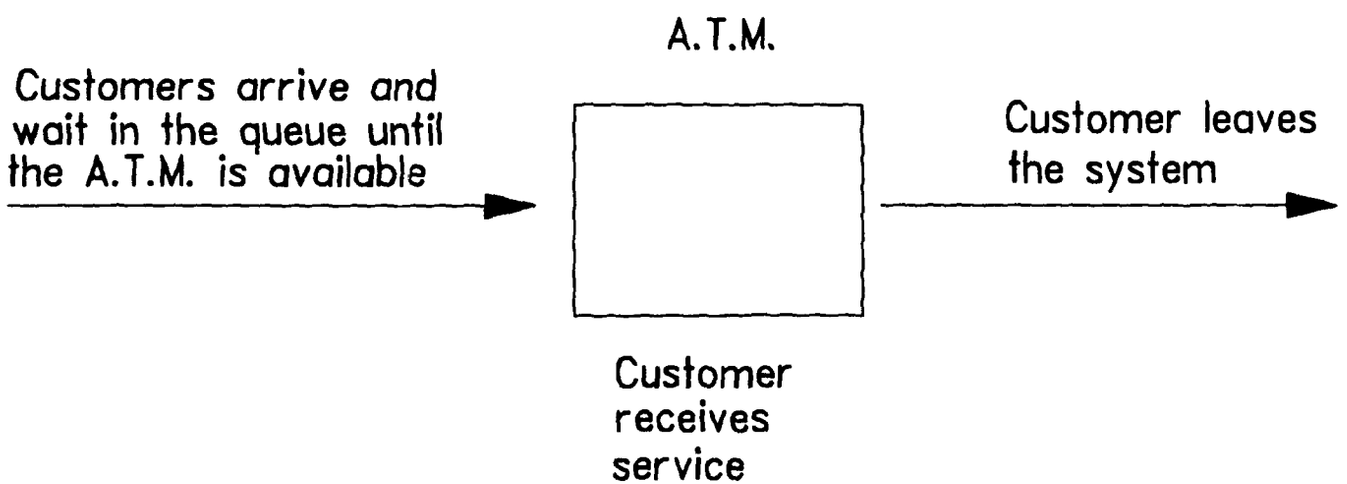

A.T.M.
Customers arrive and wait in the queue until the A.T.M. is available
Customer leaves the system
Customer receives service

Figure 1.1

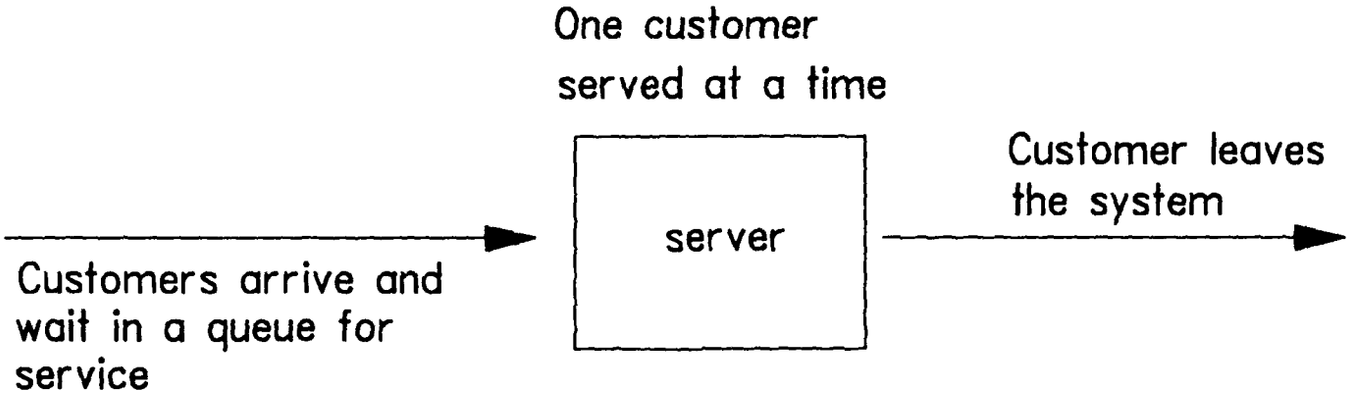

One customer
served at a time
server
Customer leaves
the system
Customers arrive and
wait in a queue for
service

Figure 2.1

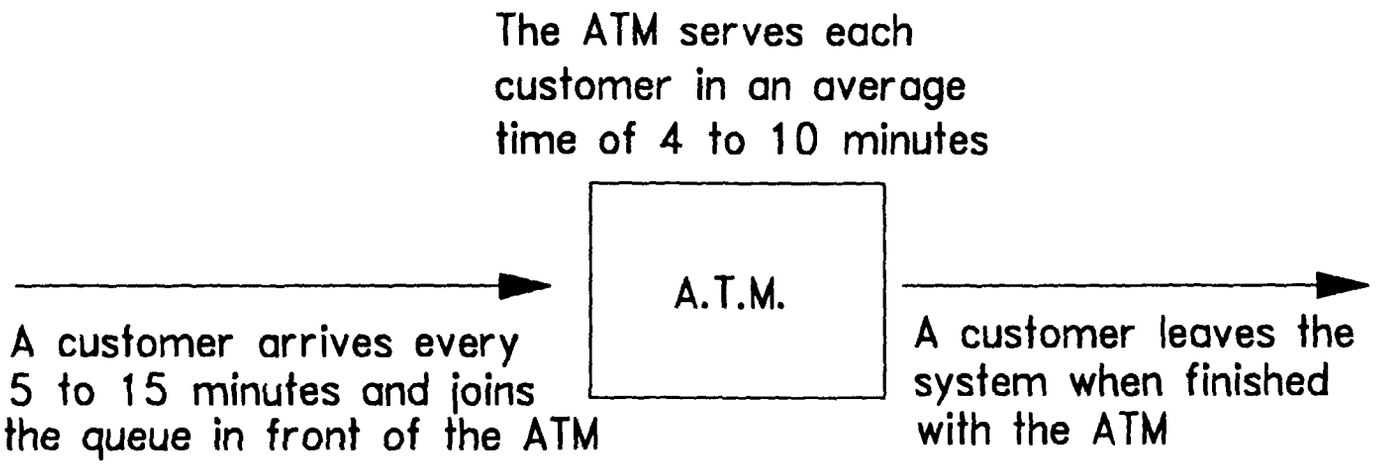
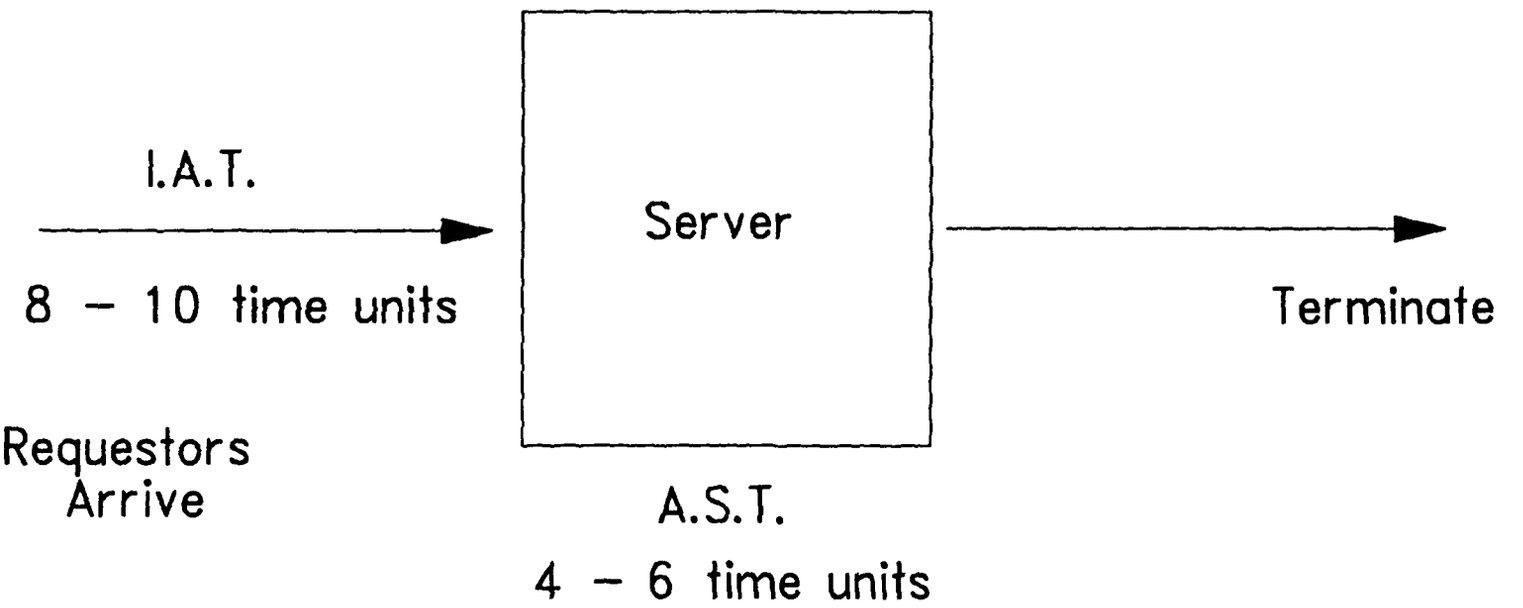

The ATM serves each customer in an average time of 4 to 10 minutes
A.T.M.
A customer arrives every 5 to 15 minutes and joins the queue in front of the ATM
A customer leaves the system when finished with the ATM

Figure 2.2

Stop simulation at time 1000



Help:
<Alt> H

Quit:
<Alt> Q

S I M U L A T I O N S E T U P

I.A.T.	Expon/ Uniform	Mean/ Min	Max
	U	8	10

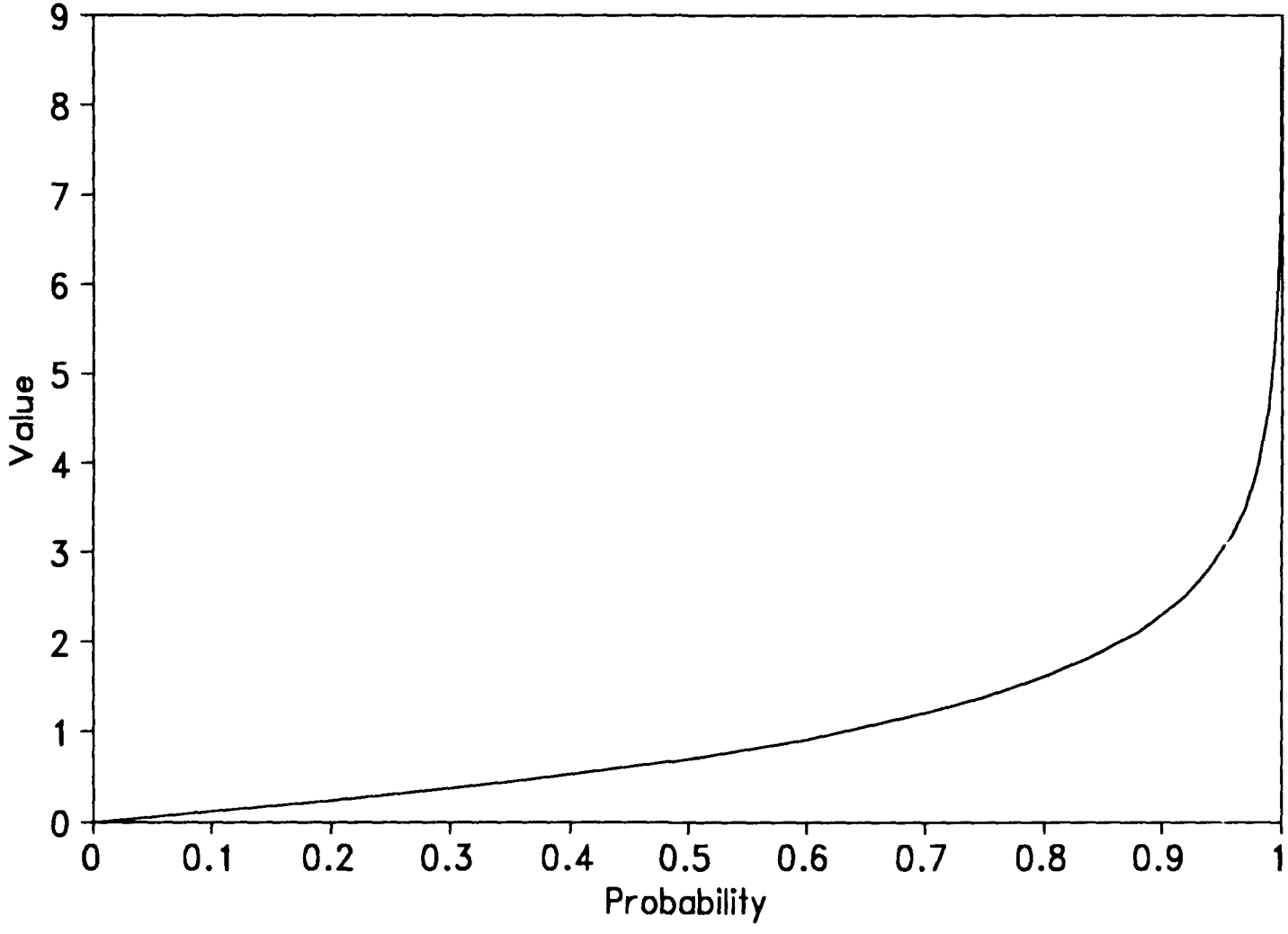
Stop at time: 1000

Stop # Req.:

Server	Expon / Uniform	A.S.T.: Min	Max	Cap	# servers reached	next server#	prob	next server#
1	U	4	6	1	0			
2								
3								
4								
5								

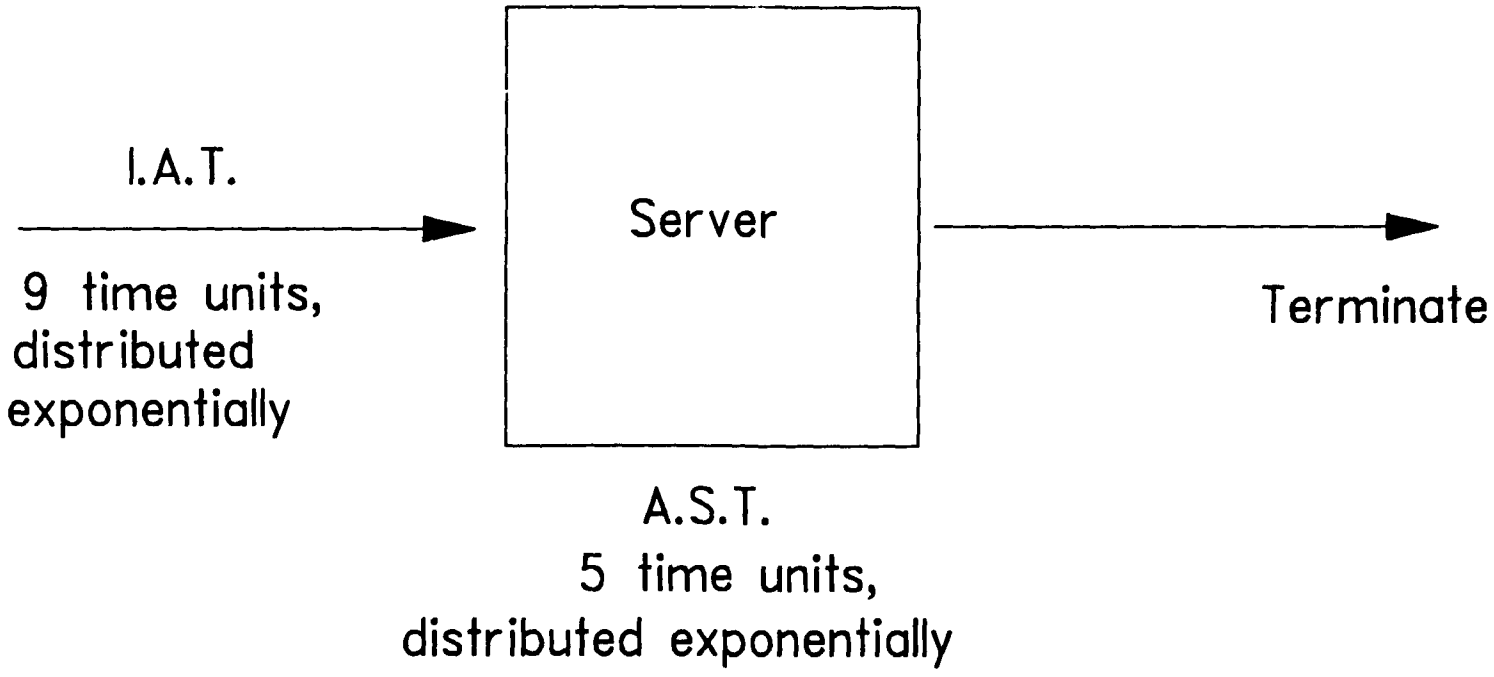
```
t  1000
u  8  10
u 4 6 1 0
```

Figure 3.2
Contents of Data File SIM.DAT

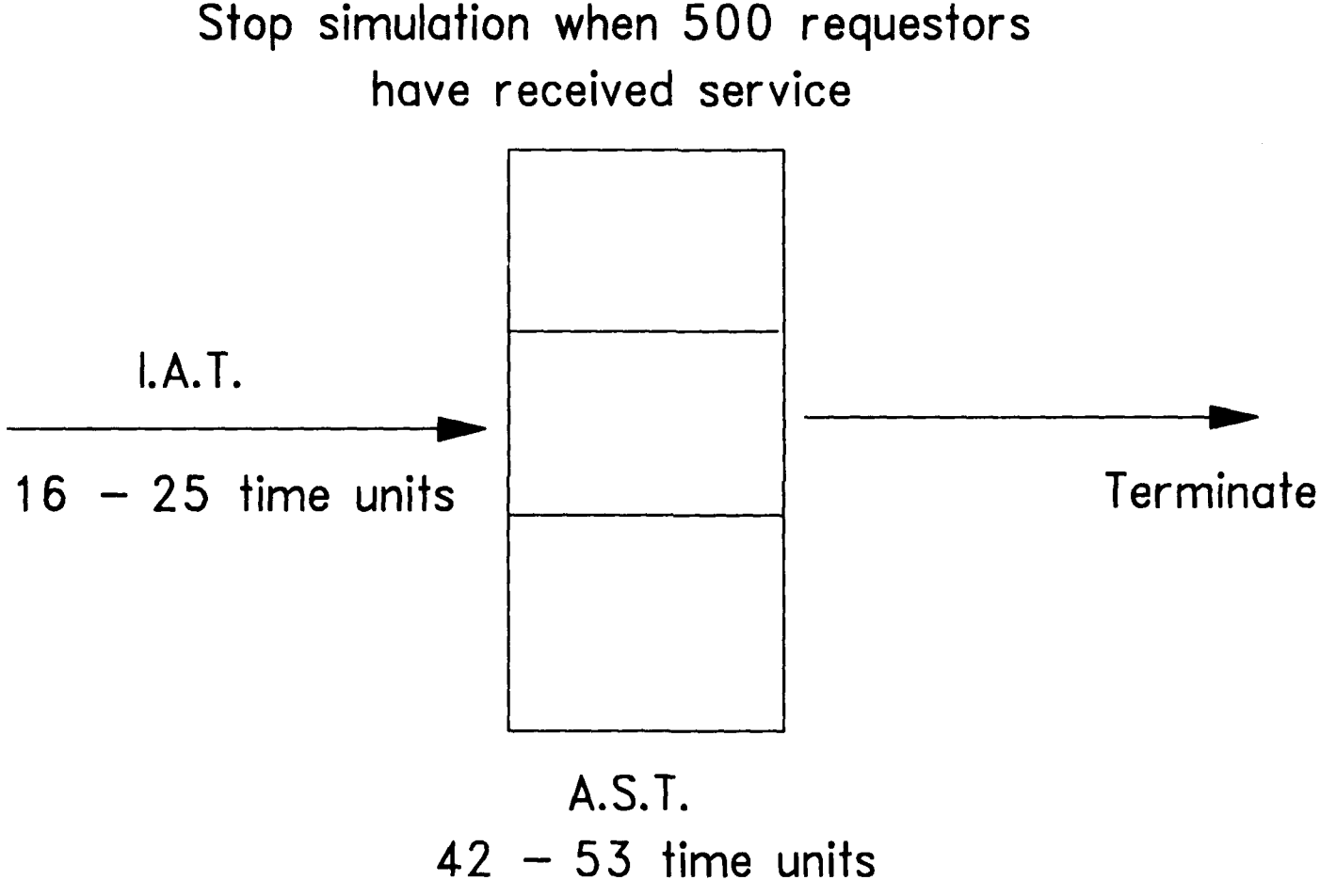


Exponential Distribution

Stop simulation at time 1000

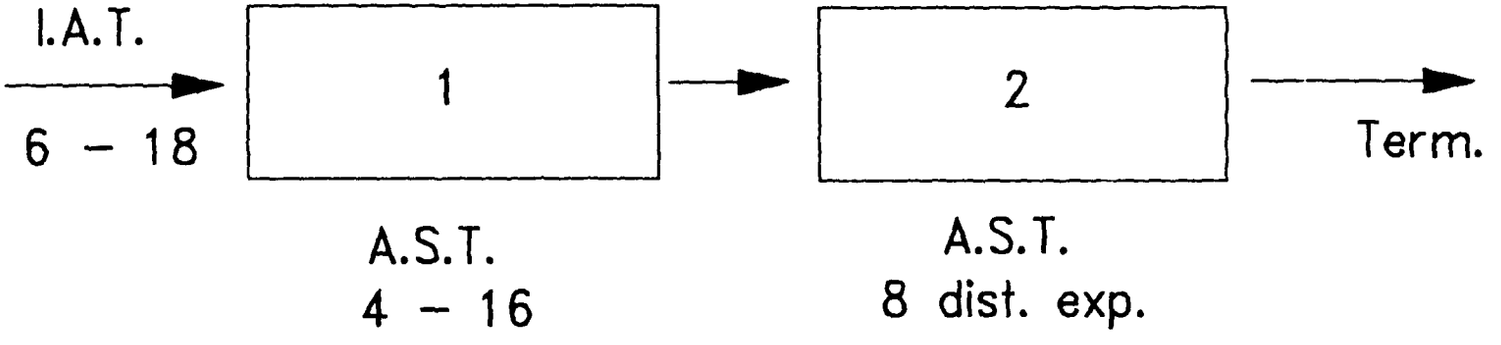


Case Study 2

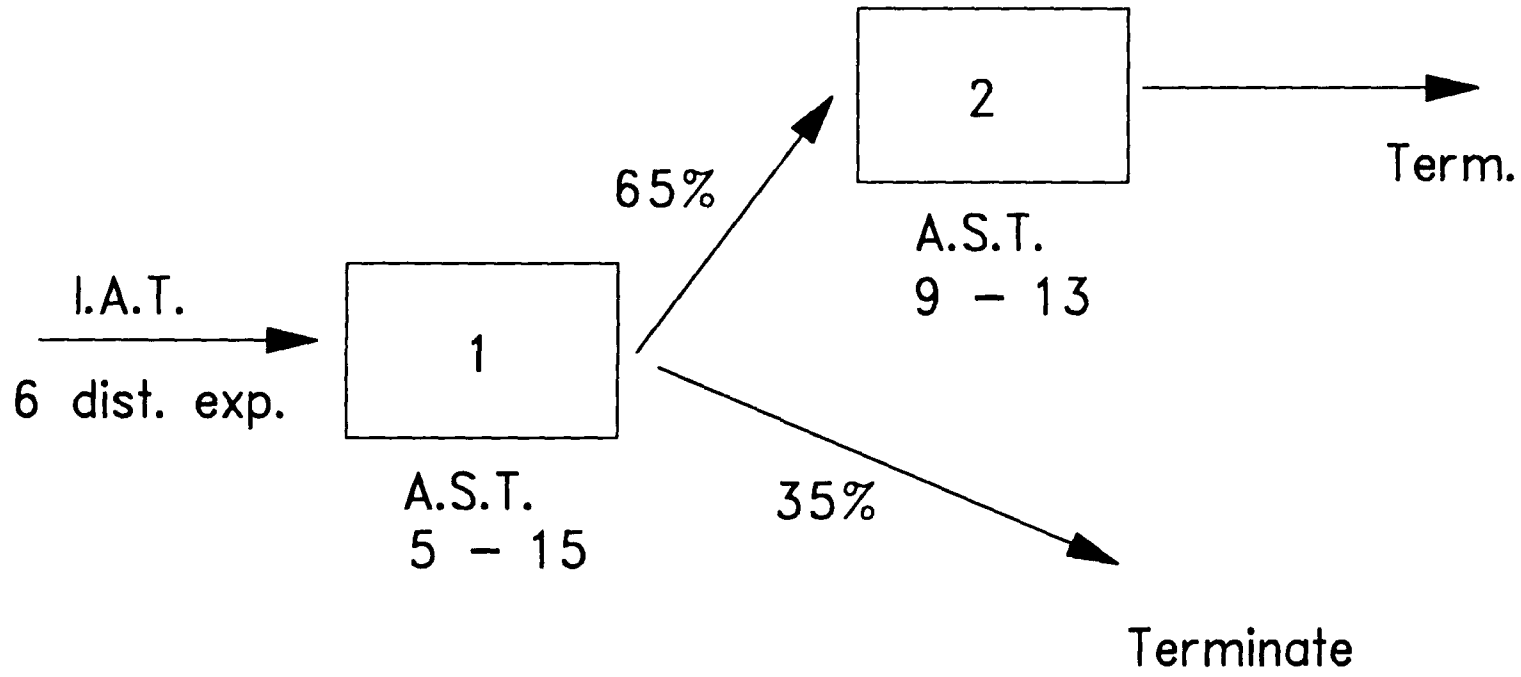

Stop simulation when 500 requestors
have received service
I.A.T.
16 – 25 time units
Terminate
A.S.T.
42 – 53 time units

Case Study 3

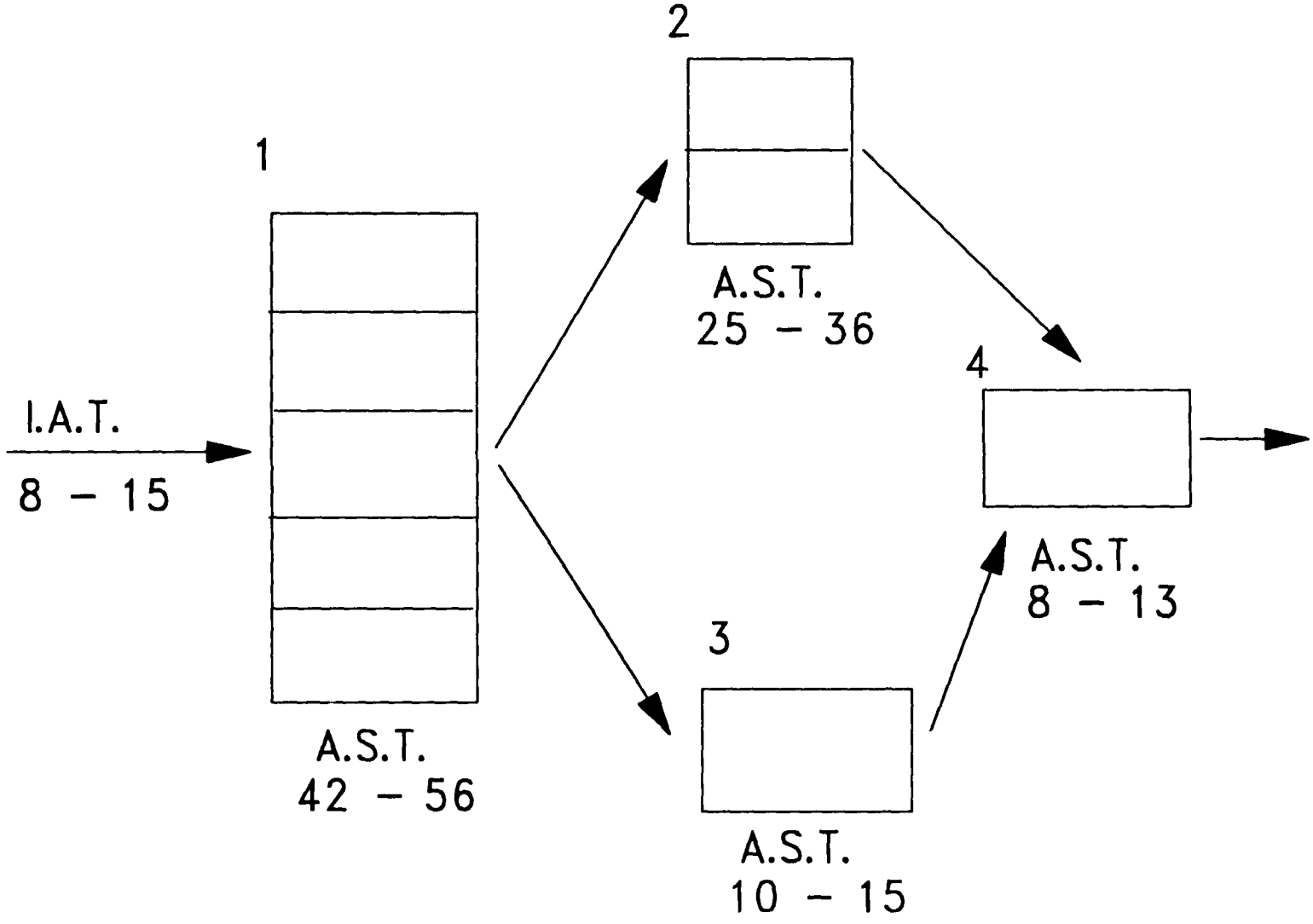
Stop simulation when 500 requestors
have received service



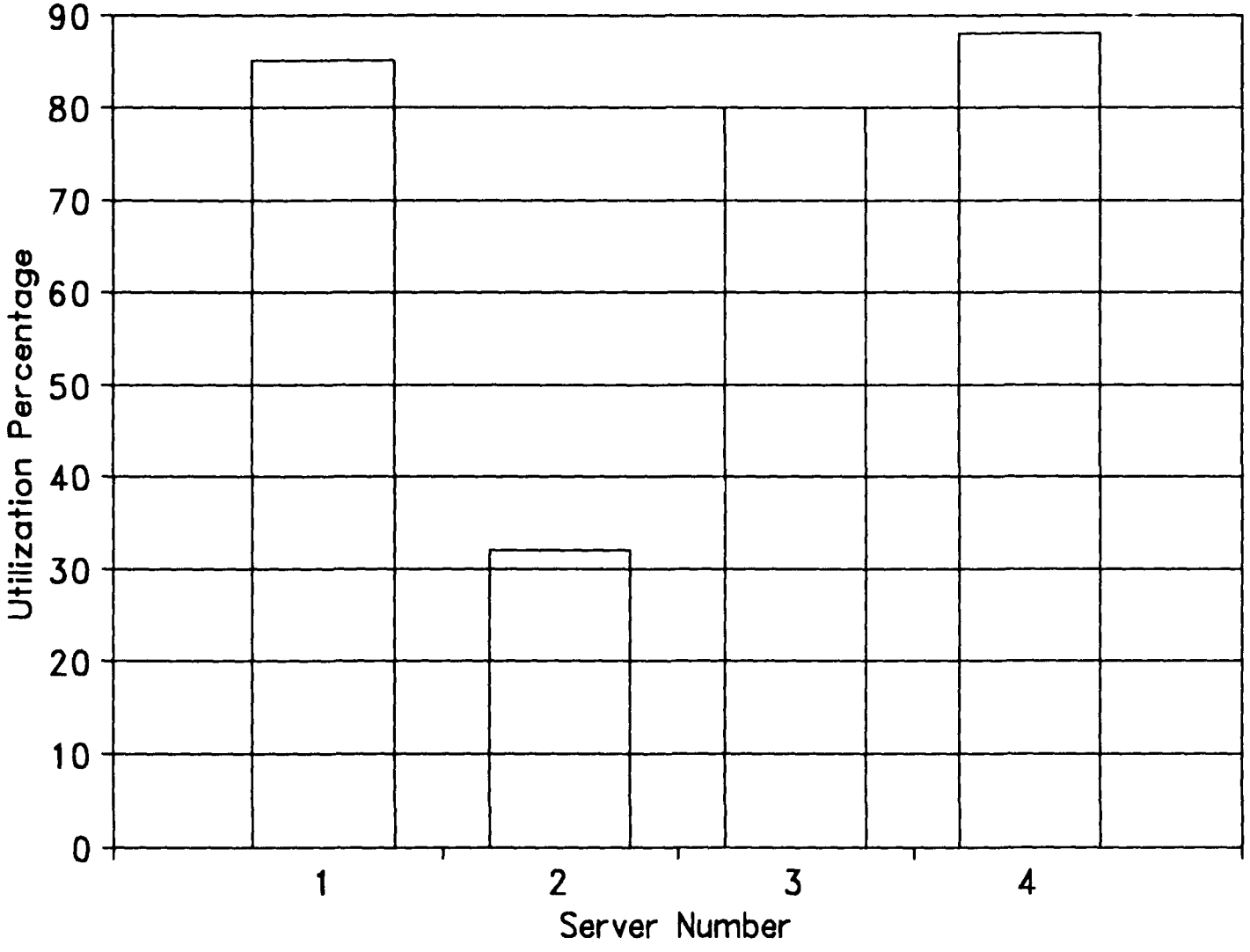
Case Study 4



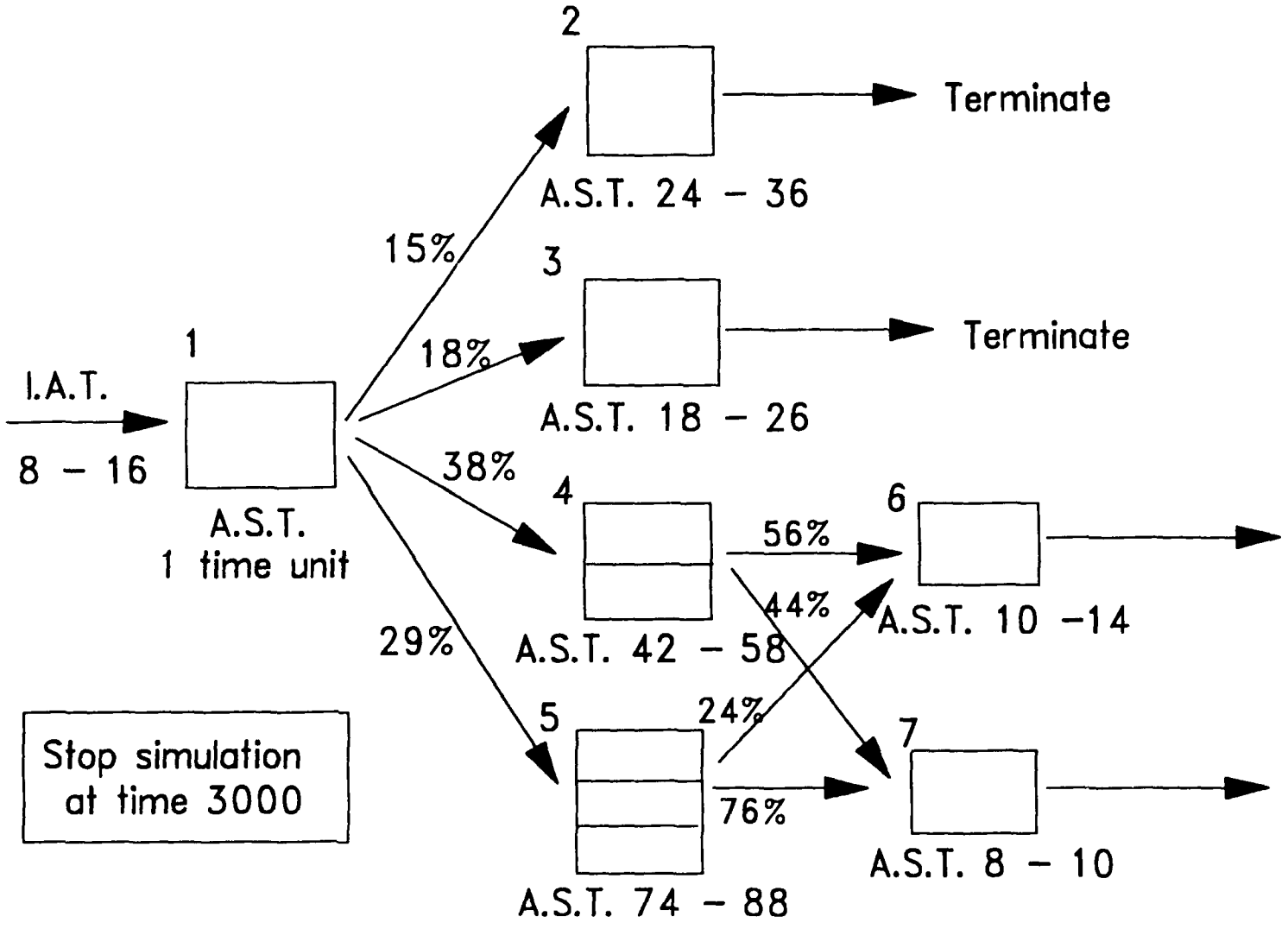
Stop simulation at time 1500



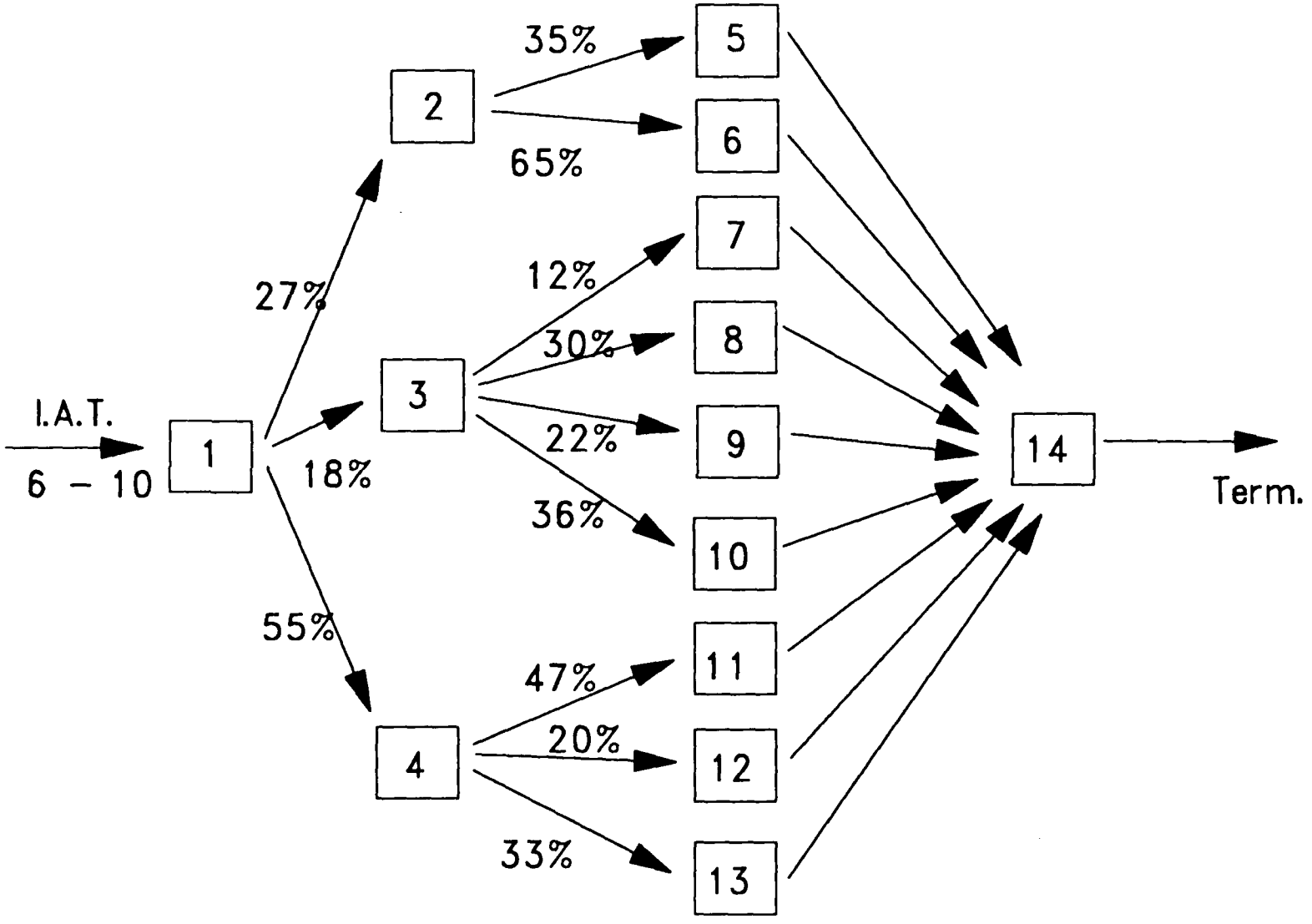
Case Study 6



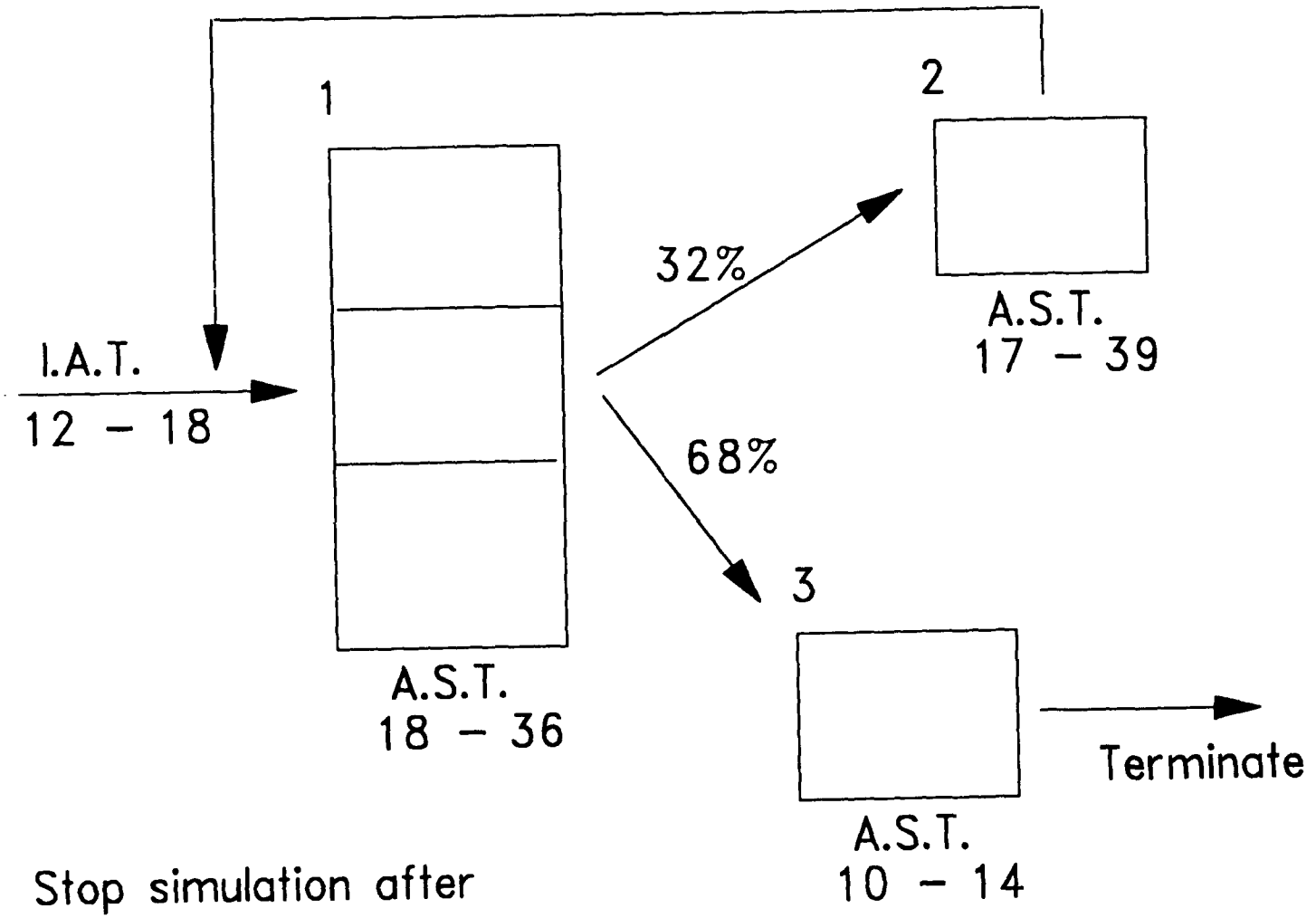
Comparison of Server Utilizations - Case Study 6


2
Terminate
A.S.T. 24 – 36
15%
3
Terminate
18%
A.S.T. 18 – 26
1
I.A.T.
8 – 16
A.S.T.
1 time unit
38%
4
6
56%
44%
A.S.T. 10 –14
A.S.T. 42 – 58
29%
24%
5
7
76%
A.S.T. 8 – 10
A.S.T. 74 – 88
Stop simulation
at time 3000

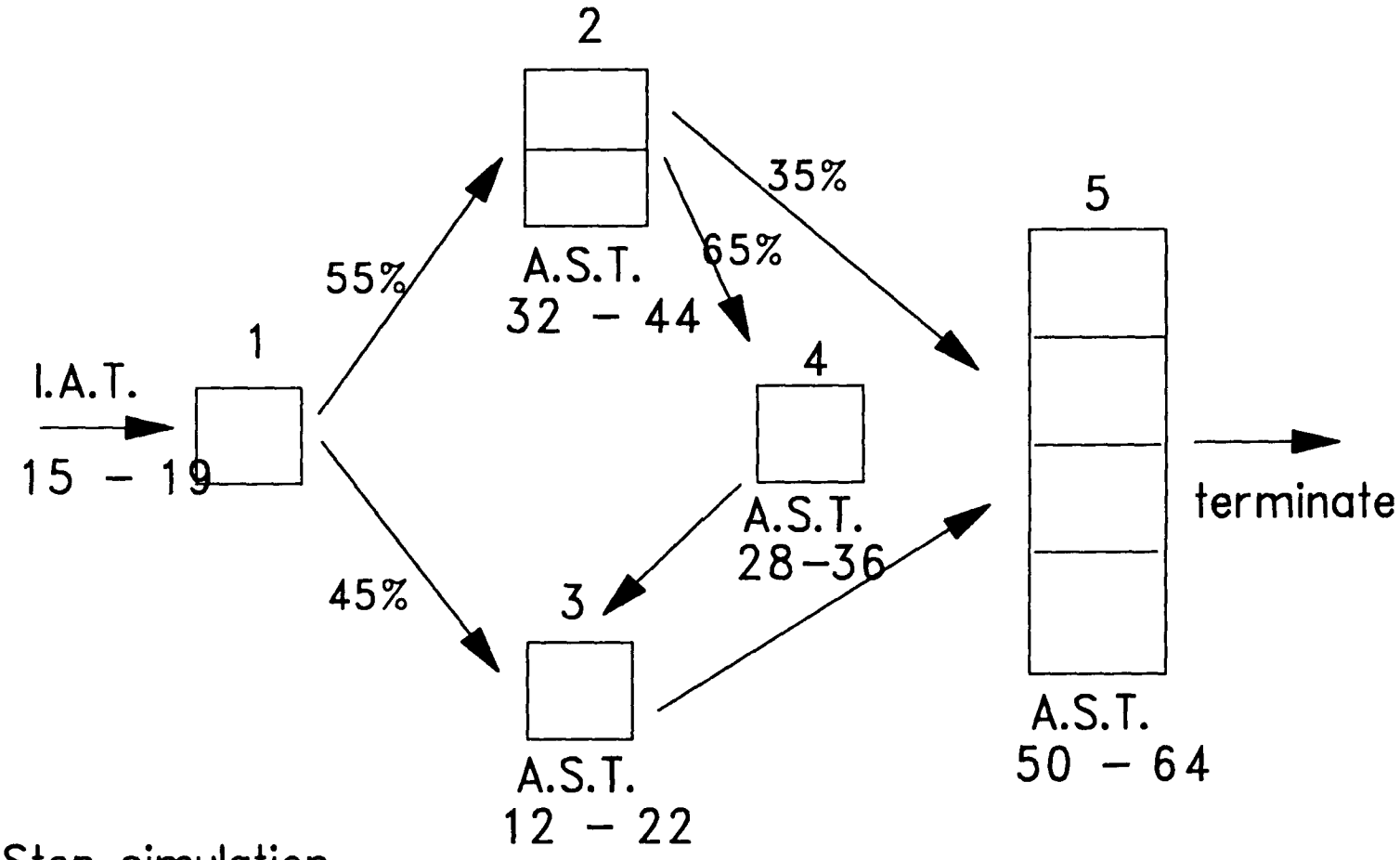
Case Study 7


I.A.T.
6 – 10
1
27%
18%
55%
2
3
4
35%
65%
12%
30%
22%
36%
47%
20%
33%
5
6
7
8
9
10
11
12
13
14
Term.

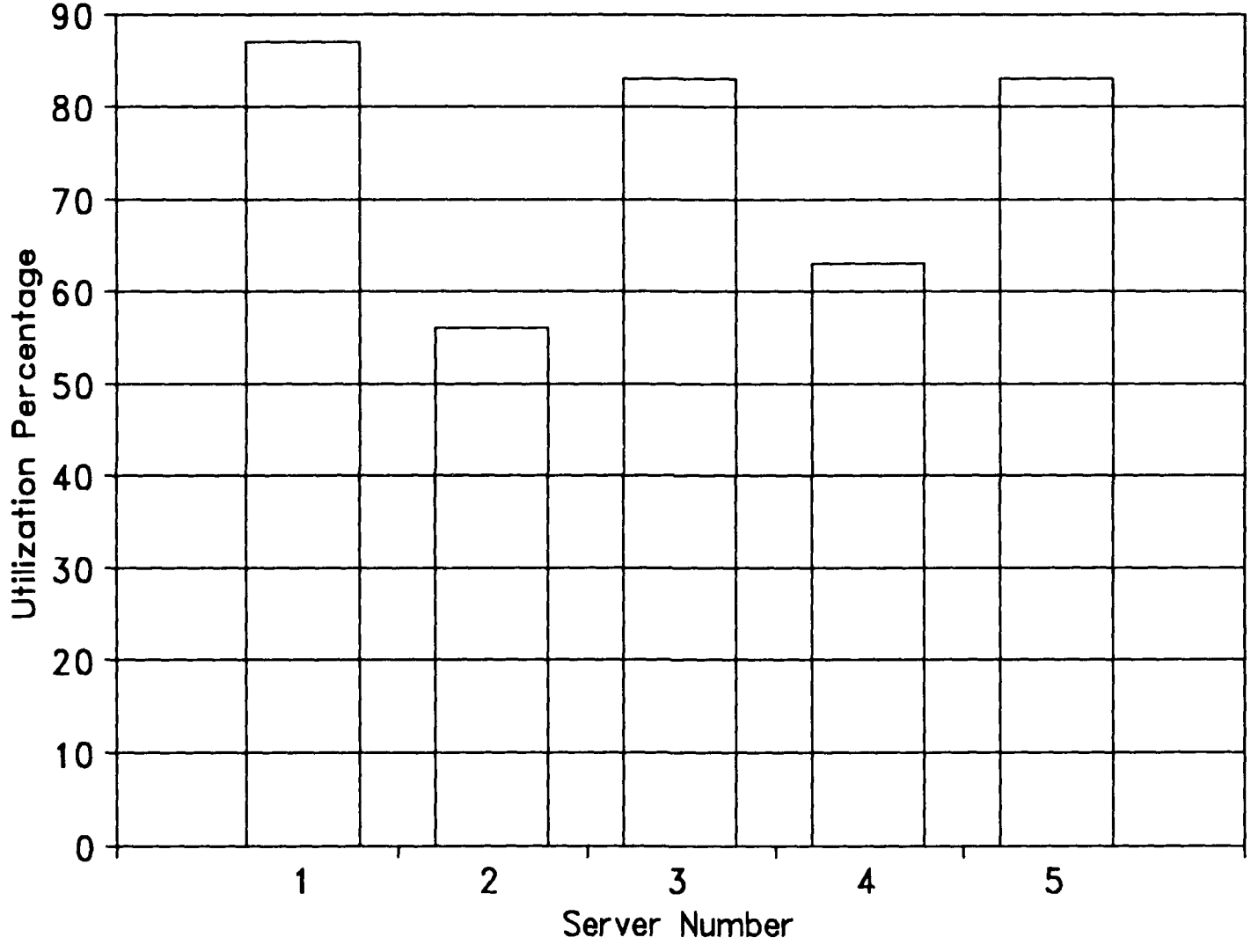
Case Study 8


1
2
3
I.A.T.
12 - 18
32%
68%
A.S.T.
17 - 39
A.S.T.
18 - 36
A.S.T.
10 - 14
Terminate
Stop simulation after
8000 time units

Case Study 9



Stop simulation
after 6000 time units



Comparison of Server Utilizations - Case Study 10

User Manual

Table of Contents

INTRODUCTION

Welcome to the SIMPACK simulation environment. SIMPACK allows you to model many types of systems on your computer, without having to know much about simulation programming and discrete-event simulations. SIMPACK automatically generates a model of your predefined system and produces a report showing various statistics and details about the behavior of the model that was simulated.

SIMPACK was developed using the Ada programming language. It provides a basic simulation environment that contains many features needed for discrete-event system simulation. The Ada source code is provided so that additional features may be added to this simulation environment.

GENERAL INFORMATION

The main purpose of SIMPACK is to simulate the behavior of a model or system that you have defined, and then produce statistical output which contains the results of that simulation. You are not required to know anything about how to program the computer to run the simulation, nor are you required to extract information from the program about the results of the simulation. SIMPACK needs basic information about the structure of your model, such as the length of time to run the simulation, the average inter-arrival time of requestors, and the average service time of the servers.

REQUIREMENTS

To be able to use SIMPACK effectively, you must have available to you the following hardware configurations:

MINIMUM CONFIGURATION
- 80386-20 MHz DOS-based computer
- 640K RAM memory
- No math-coprocessor
- Any type of monitor & keyboard
- 5.25" or 3.5" Floppy drive
- Hard disk drive with at least 1 MB free space

Having a faster CPU and a math co-processor will reduce the amount time required for simulations.

You also need the following software to be able to create and run your simulations:

- DOS 3.3 or above
- GO.BAT -- Included with SIMPACK
- RUN.EXE -- Included with SIMPACK
- SIMSETUP.SKW -- Included with SIMPACK

- SIMULATE.EXE	-- Included with SIMPACK -- Main simulation program
- SIM.DAT	-- Data file created by template

DEFINITIONS & TERMS

The following is a list of terms that are used in the setup, reporting, and description of SIMPACK:

Server: This is an entity that requestors wait for. A server holds the requestor for some amount of time to do some sort of processing, then releases the requestor. The requestor can then leave the system, or branch to another server for other processing.

Requestor: This is an entity that arrives in the system every so often (arrival time is a random number based on information supplied by you), and waits for service by a server for some amount of time. When the server is available, the requestor spends some amount of time being processed, then either exits the system, or continues on to wait for another server.

Queue: A queue (or line) forms in front of a server if there are requestors waiting to be processed by that server.

Time: Time units can be of any increment you wish, but you must be consistent in you units when specifying arrival times, service times, lengths of simulations, etc. Units can be seconds, milliseconds, nanoseconds, or even days, weeks, years, or centuries (time units are arbitrary).

Stop-time: This is the time at which you want the simulation to stop running. After this much simulation time has elapsed, stop the simulation.

Stop-req.: After this number of requestors have been processed, stop the simulation.

I.A.T.: Inter-arrival time: This is the time interval at which requestors arrive in the system. The actual arrival times are random numbers based on the following information:
If you choose a uniform probability distribution, you must supply the minimum and maximum arrival time intervals (for example, a requestor arrives every 3 to 5 minutes).
If you choose an exponential probability distribution, you must supply the mean arrival time (for example, a requestor arrives every 10 seconds, on the average).

A.S.T.: Average service time: This is the length of time that a server holds a requestor for processing. The actual service times are random numbers based on the following information:
If you choose a uniform probability distribution, you must supply the minimum and maximum service times (for example, this server spends between 5 and 15 milliseconds processing each request).
If you choose an exponential probability distribution, you must supply the mean service time (for example, this server spends 62 minutes processing each requestor, on the average).

Capacity: This is the number of requestors that a server can process all at once.

Ser. reached: This is the number of servers that requestors may branch to from the current server for other processing.

Server #: This is a unique ID # that you assign to each server in your system. This number is used for reporting on the statistics for each individual server.

Probability: This is the percentage probability that a requestor will go to a particular server after being processed by the current server. You specify probabilities when determining which servers can be reached from the current server.

Avg. Time: For server statistics, this is the average time that all requestors spent being processed by each server in the system. For queue statistics, this is the average time that all requestors spent waiting in line for service.

Entries: For server statistics, this is the number of requestors that were taken out of queue and began to be processed by a server. For queue statistics, this is the number of requestors that arrived in the queue for a server.

0-Entries: This is the number of requestors that arrived at a server, and had to wait no time in queue to begin processing.

Max contents: For server statistics, this is the maximum number of requestors that a server processed at any one time (for example, this might be 3 if the capacity of the server is 3 or more). For queue statistics, this is the maximum number of requestors in a queue at any one time.

Current cont.: For server statistics, this is the number of requestors that are still being held by a server when the simulation ends. For queue statistics, this is the number of requestors still waiting in queue when simulation ends.

Avg. contents: For server statistics, this is the average number of requestors that a server processed throughout the whole simulation (for example, this might be 1.5 if the capacity of the server is 2 or more). For queue statistics, this is the average number of requestors waiting in queue throughout the duration of the simulation (for example, requestors waited an average of 10 minutes for server 3 during the simulation).

Utilization: This is the average utilization (in percent) of a server in the system (for example, if the server was busy processing requestors only half of the time, then the utilization might be 50%).

USING SIMPACK

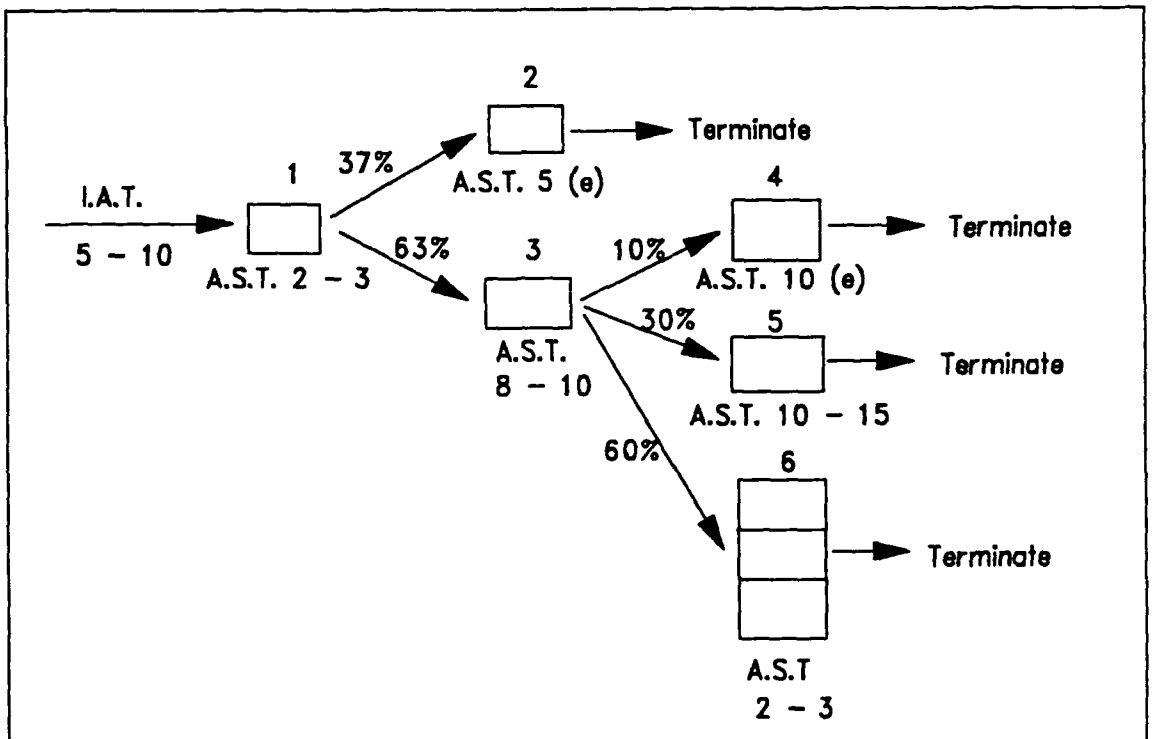
Getting Started:

To begin using SIMPACK, you must first make sure that you meet the hardware and software requirements listed in REQUIREMENTS. If you have a hard disk drive and you want to install the SIMPACK software onto it (your simulations will run much faster!), you need to copy all the files from the SIMPACK diskette onto your hard disk. Below is an example of one way you might be able to install the files (for other information about copying files and using your hard disk, please consult your DOS manual or computer system manuals):

1. Insert the SIMPACK disk in a disk drive
2. Copy all files to a subdirectory on your hard disk

Throughout the rest of this section, it is assumed that you are in your working directory, that is, the place where all of the SIMPACK files are (if you are using only a floppy drive, you should, most likely, be working from your A: or B: drive; if you are using a hard drive, you should, most likely, be working from your C: or D: drive).

The first thing you need to do is to create a complete drawing of your system, including all servers. Be sure to number each server in your picture in order, starting with 1, and do not repeat any number. Consider the following sample.

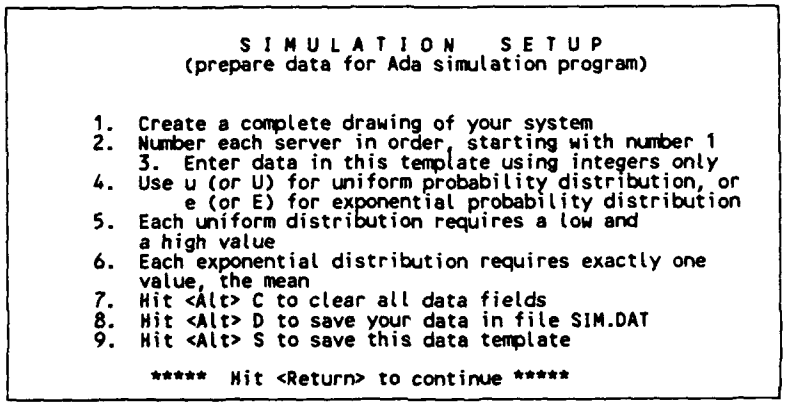


Sample of Multiple-Server System

The next thing you need to do is prepare a data file that SIMPACK will use to simulate your system. You will type "GO" to display a template that allows you to enter the necessary information for your model. When you save your information in the template, the data file, named SIM.DAT, is created. Once this file exists, you will type "SIMULATE" at the DOS prompt. This starts the simulation, using the information contained in SIM.DAT. When the simulation is complete, SIMPACK prints a statistics report for all servers and queues in your system.

Preparing Your Simulation Data File:

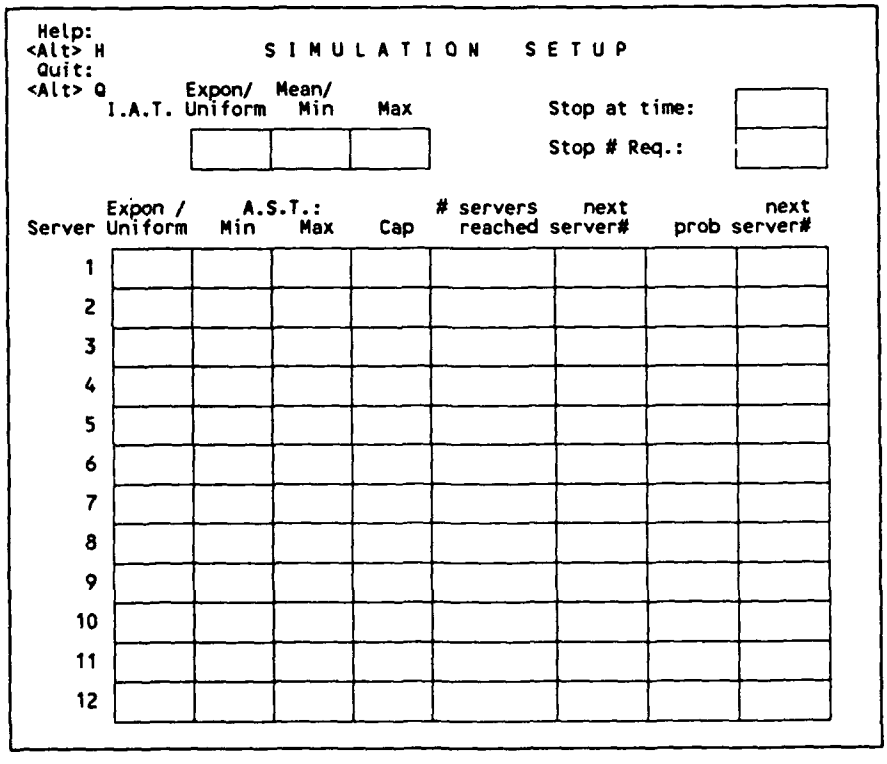
At the DOS prompt, type "GO" and press <Enter>. This brings up a template that allows you to enter the details of your model. Following are the screens that you will see:



S I M U L A T I O N S E T U P
(prepare data for Ada simulation program)

1. Create a complete drawing of your system
2. Number each server in order, starting with number 1
3. Enter data in this template using integers only
4. Use u (or U) for uniform probability distribution, or
   e (or E) for exponential probability distribution
5. Each uniform distribution requires a low and a high value
6. Each exponential distribution requires exactly one value, the mean
7. Hit <Alt> C to clear all data fields
8. Hit <Alt> D to save your data in file SIM.DAT
9. Hit <Alt> S to save this data template

***** Hit <Return> to continue *****



Help: <Alt> H
Quit: <Alt> Q

S I M U L A T I O N S E T U P

I.A.T. Expon/ Uniform | Mean/ Min | Max

Stop at time:

Stop # Req.:

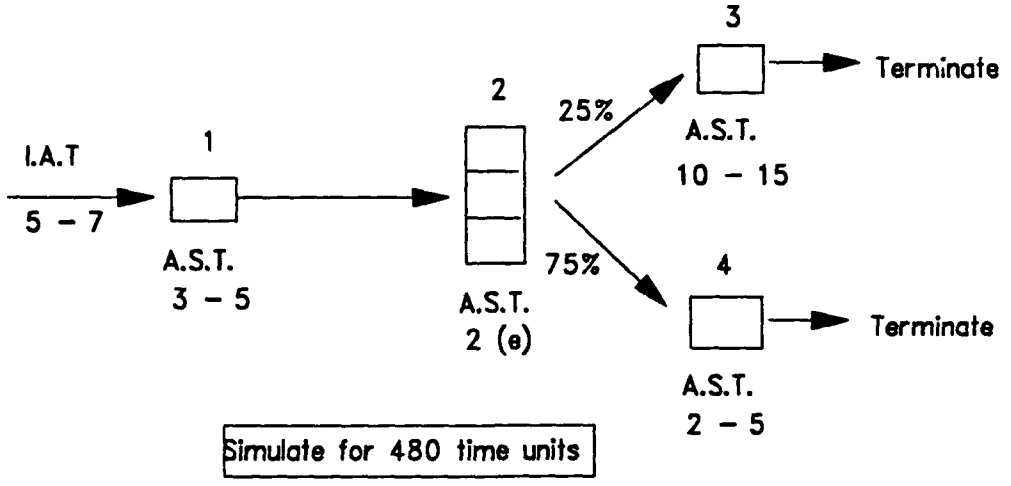
Server	Expon / Uniform	A.S.T.: Min	Max	Cap	# servers reached	next server#	prob	next server#
1								
2								
3								
4								
5								
6								
7								
8								
9								
10								
11								
12								

You may view the help screen at any time by pressing <Alt> H (hold the <Alt> key down and press the letter H). Press <Return> (or <Enter>) and you will see the main template entry above.

From here, you can press <Alt> H to display a help screen, or <ALT> Q to quit using the template and exit back to DOS (you need to make sure to save your data before exiting if you want to). If you want to save your information in the data file SIM.DAT, press <Alt> D (hold the <Alt> key down and then press the letter D). You may save the contents of the template itself by pressing <Alt> S (hold the <Alt> key down and then press the letter S).

The template allows you to define all the servers in your system, including up to 6 additional servers that can be reached by branching from any one server. The template lets you define a total of 23 different servers for your model.

As an example, consider the following model:



Sample Model

First, enter the type of probability distribution you want for arrivals to the system (I.A.T.). In this model, you should enter "u" for a uniform probability distribution (otherwise you would enter "e" for an exponential distribution). Next, enter the arrival increments: "5" for the minimum, and "7" for the maximum. This means that entities arrive in the system every 5 to 7 minutes (if you had selected "e" for an exponential distribution, you would only enter the mean arrival time, not minimum and maximum times. You would enter the mean in the "Mean/Min column, and leave the "Max" entry blank.).

Next, enter the time at which you want the simulation to end: "480" (enter this at the "Stop at time:" entry). Since you want to stop at a certain time and not after a certain number of requestors have been processed, leave the "Stop # Req.:" entry blank.

Now, you can begin defining your servers.

At "Server 1", enter "u" for uniform instead of "e" for exponential. Enter "3" for "A.S.T. Min" and enter "5" for "A.S.T. Max". Enter "1" for "Cap", since server 1 has a capacity of 1. Enter "1" for "# servers reached", since requestors can go directly to 1 other server (server 2) after finishing at this server. Enter "2" for "next server #" since server 2 is the server to branch to. Enter "100" for "prob" since all requestors leaving server 1 will go to server 2.

At "Server 2", enter "e" for exponential instead of "u" for a uniform distribution. Enter "2" for "A.S.T. Mean/Min", and leave the "A.S.T. Max" entry blank. Enter "3" for "Cap", since server 2 has a capacity of 3 (this means that server 2 can process up to 3 requestors at the same time). Enter "2" for "# servers reached", since requestors can branch to 2 other servers after finishing at this server. Enter "3" for "next server #", and enter "25" for "prob", since there is a 25% chance of branching to server 3. Keep moving right, and enter "4" for "next server #", and enter "75" for "prob", since there is a 75% chance of branching to server 4.

At "Server 3", enter "u" for uniform, "10" for "A.S.T. Min", "15" for "A.S.T. Max", "1" for "Cap", and "0" for "# servers reached", since requestors leave the system after being processed by server 3.

At "Server 4", enter "u" for uniform, "2" for "A.S.T. Min", "5" for "A.S.T. Max", "1" for "Cap", and "0" for "# servers reached", since requestors leave the system after being processed by server 4.

At this point, you have defined a simulation based on the predefined model shown above. To save this information, and create the data file SIM.DAT (which is needed to run the simulation), press <Alt> D (hold the <Alt> key down and then press the letter D). You may save the contents of the template itself by pressing <Alt> S (hold the <Alt> key down and then press the letter S). After the information is saved, you can press <Alt> Q to quit the template program and return to the DOS prompt.

Starting Your Simulation:

Once the data file (SIM.DAT) has been created by the template program, you can type "SIMPACK" at the DOS prompt to run your simulation. When the simulation is done, a statistics report will be printed to the screen.

Interpreting Your Simulation Results:

Statistics about all servers and all queues in your model are calculated and displayed after the simulation runs to completion. Server statistics include capacity, number of entries, average time per entry, maximum contents, current contents, and average contents, and utilization of every server in the model. Queue statistics include number of entries, number of entries waiting no time for service,

average time per entry in queue, maximum contents, current contents, and average contents of every queue in the system. (For explanations of these terms, you can refer to TERMS & DEFINITIONS).

For this example, the statistical output might look like the following:

```
Simulation Stopped at Time:  480
4 servers in this run.
SERVER   CAPACITY  ENTRIES  AVG. TIME PER ENTRY   MAX   CURRENT CONTENTS   AVG.   UTIL.
 1          1        77          3.9              1          0            0.6    63.1 %
 2          3        76          2.2              2          1            0.4    11.7 %
 3          1        18         13.2              1          1            0.5    49.6 %
 4          1        57          3.6              1          0            0.4    42.3 %

QUEUE   ENTRIES   0-ENTRIES   AVG. TIME     MAX.   CURRENT CONTENTS   AVG.
 1         77         77          0.0          1          0            0.0
 2         77         77          0.0          1          0            0.0
 3         19         13          2.8          2          0            0.1
 4         57         50          0.3          1          0            0.0
```

The first line of the output states that simulation stopped at time 480. The second line indicates that there were four servers in this model. The next two groupings of output are statistics for the servers and for the queues. Note that exactly one queue automatically forms in front of each server, regardless of its capacity.

Server 1 had a capacity of 1 and provided service to 77 requestors. The average time for each requestor was 3.9 time units. The server provided service to a *maximum* of one requestor at a time (recall its capacity is one). At the end of simulation, the current contents of the server was zero, i.e., the server was idle. The average contents of the server was 0.6 and the utilization of the server was 63.1%. Descriptions of the other servers is similar. Note that for server 2, its maximum contents was two, and it was processing one requestor when simulation stopped.

Queue 1 had a total of 77 entries and 77 zero entries. That is, 77 requestors spend zero time in the queue. The average amount of time requestors spent in the queue was zero, and there was a maximum of one requestor in the queue during the simulation. The average number of requestors in the queue was zero. Note that for queue 3, seven requestors spent some time in that queue and two requestors were in the queue, waiting for service, when simulation stopped.

System Manual

Table of Contents

SIMPACK is a multipurpose simulation package designed to run on a 32-bit IBM-compatible machine. It was compiled with Meridian Ada and comes with the following files:

RANDOM.ADA -- contains the random number generator package used with the program.

SIMULATE.ADA -- contains the main subprogram to drive the package.

SIMPACK.ADA -- contains the SIMPACK package, which does everything else.

To build simulate.exe, compile the files in the following order:
RANDOM.ADA, SIMPACK.ADA, SIMULATE.ADA.

This program also requires the use of an input file which will describe the system to be implemented. This file should always be named SIM.DAT.

1.0 INPUT

All input into the program is from the input file SIM.DAT. Here is an example input file:

```
t
1000
u 2 4
u 1 2 1 4 2 25 3 25 4 25 5 25
u 5 10 1 0
u 1 2 1 2 6 50 7 50
u 10 15 1 0
u 1 2 3 2 8 50 9 50
u 10 15 1 0
u 1 2 1 0
u 5 10 1 0
u 1 2 1 0
```

Figure 1 - Example Input File

Now let's dissect this file to determine what it is trying to tell us.

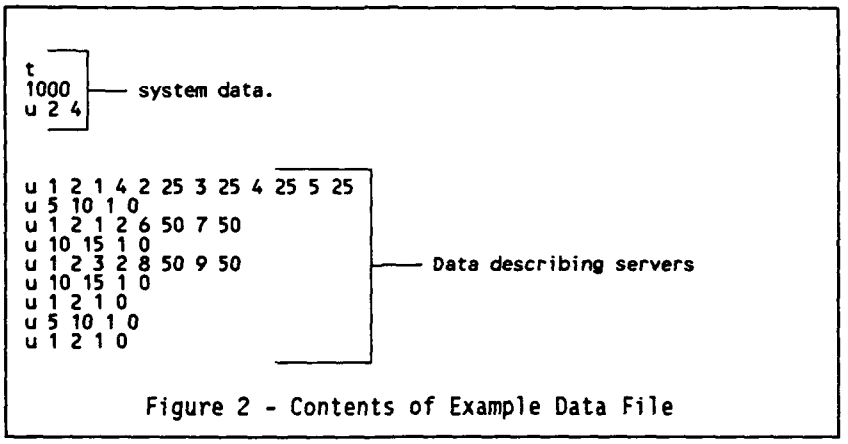


Figure 2 - Contents of Example Data File

First, let's discuss the system data.

1.1 SYSTEM DATA:

Line 1: t

The "t" tells under what conditions the simulation being specified will terminate. There are three possible letters that can appear here are 't', 'r' and 's'. 'T' stands for "time", and means that the simulation must stop after a specified time has elapsed. 'R' stands for "requestors", and signifies that the system should terminate after a number of requestors have left the simulated system. 'S' stands for "starve", and signifies that after a given time the system will stop generating new requestors. When this happens the system will process all requestors still waiting to be processed, and terminate when all requestors have left the system. In effect, the system "starves" to death. The character being read may be upper or lower case, as the program is case insensitive.

Line 2: 1000
This line specifies either a time or number of requestors, depending on what the exit condition that was just specified is. If the condition is "time" or "starve" this number is the time after which the simulation either terminates or starves. Otherwise this specifies the number of requestors the system will process before it terminates.

Of course, the system may terminate under a number of other conditions, but these are abnormal and are reached because something is going wrong with the program. Normal termination occurs because one of these three conditions has occurred.

Line 3: u 2 4
This line sets up the random number generator for the system, which needs it to generate the arrival times for new requestors. There are three pieces of information here:

'u' tells the program that the distribution for this random number generator is UNIFORM. The only other possibility at this time is 'e' for EXPONENTIAL.

"2 4" specifies the range of values (minimum .. maximum) that the random number generator may generate. In this case, the random number may be no less than 2 and no greater than 4.

If distribution is EXPONENTIAL, we cannot specify a range of values. Instead, we specify a mean and the program will generate random number whose mean will (hopefully) be that specified. Of course, if we need a mean and not a range we only need one number. So an exponential specification would look like this:

"e 3"

All system data is read in by SIMULATE, and some of it is passed in as a parameter to START_SIMULATION, which then sets the appropriate fields in the system variable SYS. (The reason for this setup is that SYS is not visible from SIMULATE because it is outside of the package SYS is in, SIMPACK. Thus, if we want to change SYS's fields we have to do it from a procedure inside the SIMPACK package. So we pass the system data we wish to set as a parameter to START_SIMULATION, which changes the global variable).

That just about covers the first three lines of the input file. What about the server data?

1.2 DATA DESCRIBING SERVERS.

There are nine lines (see figure 2) describing servers, each line describing a single server. There is no reason the data all has to be on one line, but I find this format to be more readable than other forms. Here is an example line describing server #5:

u 1 2 3 2 8 50 9 50

Now let's cut this line apart to find out what it is telling us.

"u 1 2" This describes the random number generator in exactly the same way that we described the random number generator (RNG) in the system data. The difference is that the system RNG is used to tell how often arrival times occur, while THIS RNG is used to generate the service time when a requestor is processed. As before, this could be made an EXPONENTIAL-distribution RNG by giving the letter E and a mean such as "e 100." THIS WILL PROBABLY BE THE SOURCE OF A LOT OF BUGS SO PAY ATTENTION: Remember that when an exponential distribution is asked for it only needs ONE number, not the TWO needed for uniform distribution. If a user enters "e 3 7" -- an easy mistake to make since we have so many numbers on this line -- things will get weird.

"3" Sets the capacity of the server to 3.

"2 8 50 9 50" This tells it that TWO servers can be reached from this server, that one of them is server 8 and there is a 50 % chance it will go there. The other is server 9 and that this server should send a requestor here if that requestor doesn't go to server 8.

The template for this line is

| Distribution type | Min/Mean service time | Max service time | Capacity
(u e) (UNIFORM ONLY)

| Number of servers pointed to | 1st reachable server | probability that the requestor will go there

etc...

"What if I don't want the requestor to go ANYWHERE from this server? What if I just want it to terminate?"

In that case, Put a 0 (zero) in the "number of servers pointed to" field and make that the end of the line. Here is an example:

"e 10 1 0"

Which means that this server has an exponential distribution with mean 10, a capacity of one, and any requestor leaving it must leave the system.

"What if I want a requestor arriving at this server to either leave the system or be redirected to another server?"

Well, this is the way the probabilities work to determine where a server goes next:

*** 1. A random number that is greater than or equal to 1 and less than 100 (the minimum might be zero .. I'm not sure).

2. Add the first probability on the list to SUM, which is initialized to zero. Compare the test value to SUM, and if TEST < SUM send it to the first server. Else move onto the next pointer and repeat step 2 until the test value is lower than the sum. If the test value is higher than all the probabilities added together, the requestor is made to exit.

   Thus, the way to guarantee that a requestor will DEFINITELY go to another server from the current one is to make certain that the probabilities of the requestor sum to 100. If the probabilities do not sum to 100, then there is a (100 - sum of probs.) % chance that the requestor will terminate. For example, the line

   "e 45 2 1 8 50"

tells us that this server has an exponential distribution with a mean of 45, that the server has a capacity of 2, that one server can be reached from this one, that the one server that can be reached from here is server 8 and that there is a 50% chance it will go there. This means that there is a (100 - 50) = 50 % chance that a requestor leaving this server will terminate.

Here is the code for the function called SET_DESTINATION that performs this task:

```
-- This function determines which server the requestor REQ should visit
-- next, based on the possible servers that can be reached from SERVER
-- and the probability distribution describing how probable it is
-- that the requestor will visit a given server. This information is held in
-- a linked list pointed to by the current server's DISTRIBUTION field.
-- The function returns the identification number of the next
-- server to be visited. If an exception is raised the value zero is returned.
-- Note that when the requestors WHERE_NEXT field := 0 it means that the
-- requestor should leave the system next.  There is no server 0.
--

function SET_DESTINATION (REQ: REQUESTLINK; SERVER: SERVER_LIST_TYPE)
                         return INTEGER is
TEST_LIST: DISTRIBUTION_LIST;
TEST, SUM, COUNT, ID: INTEGER := 0;
DESTINATION_FOUND: BOOLEAN := FALSE;
begin
TEST_LIST := SERVER.DISTRIBUTION;
if SERVER.NUM_PATHS = 0 then
   return 0;
else
TEST := RANDOM_INT (100);
COUNT := 1;

while not DESTINATION_FOUND loop
      SUM := SUM + TEST_LIST.PROBABILITY;
      if TEST <= SUM then
         ID := TEST_LIST.WHERE_TO_GO;
         DESTINATION_FOUND := TRUE;
      end if;
      if not DESTINATION_FOUND then
         COUNT := COUNT + 1;
         TEST_LIST := TEST_LIST.NEXT;
      end if;
      if COUNT > SERVER.NUM_PATHS then
         DESTINATION_FOUND := TRUE;
         ID := 0;
      end if;
      end loop;
return ID;
end if;
exception
when others =>
     PUT ("SET_DESTINATION has failed.  Zero will be returned. ");
     NEW_LINE;
     return 0;
end SET_DESTINATION;
```

Figure 3 - Listing of Function SET_DESTINATION

Before we leave, here are some more example inputs:

```
    "u 5 10 1 1 1 100"
```

This server has a uniform distribution with range 5..10. It has capacity 1 and can reach one other server: Server #1, and a requestor will always go there.

```
    "u 13 20 4 0"
```

This server has a uniform distribution with range 13..20. It has a capacity of 4 and no servers can be reached from here. This means that a requestor arriving here must exit the system when it leaves this server.

"e 1 2 3 2 25 3 25 4 25"

This server has an exponential distribution with a mean of one. It has a CAPACITY of 2 and three servers may be reached from here: a 25% chance to reach server 2. a 25% chance to reach server 3 and a 25% chance to reach server 4. Thus, there is a 75% chance that a requestor leaving this server will be redirected to another server, and a 25% chance that it will exit.

1.3 OPERATIONS ON THE INPUT FILE:

The input file must always be named SIM.DAT and is operated on by two procedures: CREATE_SERVER (in SIMPACK.ADA) and SIMULATE (in SIMULATE.ADA). SIMULATE opens the file and reads in the SYSTEM DATA. CREATE_SERVER then reads in the data DESCRIBING SERVERS. SIMULATE then closes the data file.

The experienced Ada programmers out there will recognize a minor difficulty: the input file contains both numbers and characters, and in Ada everything in a file must be of the same type! We solve this by treating the input file as if it consisted entirely of characters, and we have a function called INTGET which converts the character strings read into integers. Thus, CREATE_SERVER and SIMULATE don't **REALLY** read anything. Instead, they call our two input functions INTGET and CHARGET, which get an integer or character respectively and return the data object to the calling unit.

A feature of INTGET and CHARGET is that they have attitudes. By this I mean that INTGET is very touchy about the data it will accept. It will read an integer string (such as "234") and return it as 234, ignore whitespace and keep reading if it sees it, and abort the entire program if it reads an alphabetic or punctuation character. Similarly, CHARGET accepts a single character, ignores whitespace and keeps reading if it sees it, and kills the program if it sees a numeric character.

The reasoning behind this unfriendly behavior is that INTGET is only used when we are sure the next string should be numeric. CHARGET is also only used if we are sure the next character will be alphabetic. Thus, if INTGET sees a character or CHARGET a number, that means one of two things has happened:

1. We have somehow missed the data we were supposed to read in (which has happened occasionally).

2. The input file is improperly formatted.

In either case, the integrity of the data being read in is compromised and to continue would produce wild results. Thus, the simulation is aborted.

Figure 4 contains the code for the four procedures that access the input file SIM.DAT. Following is a listing of procedures CHARGET, INTGET, CREATE_SERVER (in SIMPACK.ADA) and SIMULATE (in SIMULATE.ADA).

```
-------------------------------------------------
--
--  This function pulls a character out of the file described by
--  FILE and returns it.
--  Note that it expects to read a letter a .. z or A .. Z.  If some
--  other visible character is read (such as a number or punctuation mark)
--  the function will not return anything but instead propagate the
--  PROGRAM_ERROR exception to the calling unit.  PROGRAM_ERROR will also
--  be propagated to the calling unit if the end of the file is reached before
--  a proper character is read.
--
function CHARGET (FILE: TEXT_IO.FILE_TYPE) return CHARACTER is
INPUT: CHARACTER := ASCII.BEL;
END_FILE, BAD_DATA: EXCEPTION;
begin
while CHARACTER' POS (INPUT) < 33  loop
      if TEXT_IO.END_OF_FILE (FILE) then
         raise END_FILE;
      else
         TEXT_IO.GET (FILE, INPUT);
      end if;
end loop;
if (CHARACTER' POS (INPUT) > 64 and CHARACTER' POS (INPUT) < 91)
   or (CHARACTER' POS (INPUT) > 96 and CHARACTER' POS (INPUT) < 123)
then
   return INPUT;
else raise BAD_DATA;
end if;
exception
when END_FILE =>
     PUT ("I tried to read a character but could not find it before");
     NEW_LINE;
     PUT ("I reached the end of the file."); NEW_LINE;
     raise PROGRAM_ERROR;
when BAD_DATA =>
     PUT ("An invalid character input has been entered."); NEW_LINE;
     PUT ("ASCII value of character is"); PUT (CHARACTER'POS (INPUT)); NEW_LINE;
     raise PROGRAM_ERROR;
end CHARGET;
-------------------------------------------------
--
-- This function gets an array of characters from the
-- text file FILE, converts it into an integer number
-- and returns it. Example: the string "234" is returned as
-- the integer number 234.
-- This function expects to read a number of some kind, and if a visible (
-- i.e. non-whitespace) character that is not in 0 ..9 is read, the exception
-- PROGRAM_ERROR will be propagated to the calling unit.  This will also happen
-- if the end of file is reached before a number is read.
-- This simulation program is designed to use 16-bit arithmetic on a 32-bit
-- machine.  Thus, a number larger than 32767 will cause NUMERIC_ERROR to be
-- propagated to the calling environment.
--

function INTGET (FILE: TEXT_IO.FILE_TYPE) return INTEGER is
CHAR_ARRAY: array (1..80) of CHARACTER;
MULTIPLIER: INTEGER := 1;
COUNTER: INTEGER := 1;
STOP :BOOLEAN := FALSE;
DUMMY: CHARACTER := ASCII.BEL;
TEMP: INTEGER;
```

```
RETURN_VALUE: INTEGER := 0;
BIG1, BIG2: INTEGER_32;
END_FILE, CHARACTER_INPUT: EXCEPTION;
begin

while (not (TEXT_IO.END_OF_FILE(FILE))) and
      (not (CHARACTER' POS (DUMMY) > 47 and CHARACTER'POS (DUMMY) < 58))
      loop
      TEXT_IO.GET (FILE, DUMMY);
      if (CHARACTER' POS (DUMMY) > 32 and CHARACTER' POS (DUMMY) < 48 )
      or CHARACTER' POS (DUMMY) > 57 then
         raise CHARACTER_INPUT;
      end if;

end loop;

if (not (CHARACTER' POS (DUMMY) > 47 and CHARACTER' POS (DUMMY) < 58))
   and TEXT_IO.END_OF_FILE (FILE) then
   raise END_FILE;
end if;

CHAR_ARRAY (COUNTER) := DUMMY;
COUNTER := COUNTER + 1;
while not STOP loop
         if TEXT_IO.END_OF_LINE (FILE) or TEXT_IO.END_OF_FILE (FILE) then
            STOP := TRUE;
         else
            TEXT_IO.GET (FILE, DUMMY);

            if TEXT_IO.END_OF_LINE (FILE) or TEXT_IO.END_OF_FILE (FILE) then
               STOP := TRUE;
            end if;
            if CHARACTER'POS (DUMMY) > 47 and CHARACTER'POS (DUMMY) < 58 then
               CHAR_ARRAY(COUNTER) := DUMMY;
               COUNTER := COUNTER + 1;
               else STOP := TRUE;
               end if;
         end if; -- if we can read a character.
end loop;
COUNTER := COUNTER - 1;

while COUNTER >= 1 loop
      TEMP := (CHARACTER' POS (CHAR_ARRAY (COUNTER))) - 48;
      TEMP := TEMP * MULTIPLIER;
      BIG1 := INTEGER_32 (RETURN_VALUE);
      BIG2 := INTEGER_32 (TEMP);
      if BIG1+BIG2 > 2 ** 15 -1 then
         raise NUMERIC_ERROR;
      end if;
      RETURN_VALUE := RETURN_VALUE + TEMP;
      if MULTIPLIER < 10000 then
         MULTIPLIER := MULTIPLIER * 10;
      elsif COUNTER > 1 then
            raise NUMERIC_ERROR;
      end if;
      COUNTER := COUNTER - 1;
end loop;

return RETURN_VALUE;
exception
when CHARACTER_INPUT =>
```

```
     PUT ("I read a character where I expected a number.");  NEW_LINE;
     PUT ("There must be something wrong with the input file."); NEW_LINE;
     raise PROGRAM_ERROR;
when END_FILE =>
     PUT ("I have reached the end of the input file and I could not");
     NEW_LINE;
     PUT ("Find an integer value I could accept."); NEW_LINE;
     raise PROGRAM_ERROR;
when NUMERIC_ERROR =>
     PUT ("This simulation package only allows user values of 16 bits.");
     NEW_LINE;
     PUT ("In other words, the maximum value that can be given is 32767.");
     NEW_LINE;
     raise NUMERIC_ERROR;
end INTGET;
-------------------------------------------------
--
-- This procedure creates a server, specifying the minimum service time,
-- the maximum service time, and the maximum capacity of the server.
-- Having done this, it inserts the server onto SERV_LIST, which is
-- a global variable and a SIDE EFFECT.  Note that this server must read
-- INPUT_FILE to get much of it's data. This file (named SIM.DAT) must be in
-- the current working directory.
-- In event of an exception, PROGRAM_ERROR is propagated to the calling
-- environment.
-- CREATE_SERVER will also check various data to see if it is accurate or not.
-- CREATE_SERVER will fail under the following conditions:
-- 1. If capacity is less than one for any server.
-- 2. If the character specifying the distribution is not 'u', 'U', 'e'
--    or 'e'.
-- 3. If any of the associated service times (minimum, maximum, or in
--    the case of EXPONENTIAL distribution mean, which is stored in the
--    minimum field ASTLOW) is less than 1.
-- 4. If the sum of the probabilities is greater than 100.
--
-- Note that if the minimum value is greater than the maximum value, they
-- will be flipped.

procedure CREATE_SERVER (HIGHEST_SERVER, SERVER_COUNT: out INTEGER;
                         INPUT_FILE: in TEXT_IO.FILE_TYPE) is
LOW, HIGH, CAPACITY: INTEGER := -9000;
COUNTER: INTEGER;
NEWITEM: SERVER_LIST_TYPE;
TEMP: DISTRIBUTION_LIST;
WHAT_KIND: PROBABILITY_DISTRIBUTION;
ANSWER: CHARACTER := 'a';
LOCAL_ASTLOW, LOCAL_ASTHIGH: INTEGER; -- Local ASTLOW and ASTHIGH.
TOTAL_PROBABILITY: INTEGER := 0;
begin
if SYS.HIGHEST_SERVER = 0 then
   SYS.HIGHEST_SERVER := 1;
end if;
while ANSWER /= 'u' and ANSWER /= 'U' and ANSWER /= 'e' and ANSWER /= 'E' loop
 ANSWER := CHARGET (INPUT_FILE);
end loop;
if ANSWER = 'u' or ANSWER = 'U' then
   WHAT_KIND := UNIFORM;
else WHAT_KIND := EXPONENTIAL;
end if;

if WHAT_KIND = EXPONENTIAL then
```

```
  LOW := INTGET (INPUT_FILE);
  if LOW < 1 then
     PUT ("Associated service times of less than one are not permitted.");
     NEW_LINE;
     raise PROGRAM_ERROR;
  end if;
  HIGH := 2 ** 15 - 1;

else
  LOW := INTGET (INPUT_FILE);
  if LOW < 1 then
     PUT ("Associated service times of less than one are not permitted.");
     NEW_LINE;
     raise PROGRAM_ERROR;
  end if;
  HIGH := INTGET (INPUT_FILE);

  if HIGH < 1 then
     PUT ("Associated service times of less than one are not permitted.");
     NEW_LINE;
     raise PROGRAM_ERROR;
  end if;
end if;

      CAPACITY := INTGET (INPUT_FILE);
      if CAPACITY < 1 then
         PUT ("Capacity must be greater than zero."); NEW_LINE;
         raise PROGRAM_ERROR;
      end if;
if LOW > HIGH then
   PUT ("The entered minimum service time is greater than the maximum.");
   NEW_LINE;
   PUT ("I shall use the lower of the two values as the minimum and ");
   NEW_LINE;
   PUT ("the other as the maximum service time."); NEW_LINE;
   LOCAL_ASTLOW := HIGH;
   LOCAL_ASTHIGH := LOW;
else
   LOCAL_ASTLOW := LOW;
   LOCAL_ASTHIGH := HIGH;
end if;

NEWITEM := new SERVER_TYPE;

NEWITEM.NUM_PATHS := INTGET (INPUT_FILE);
-- PUT ("Number of paths = "); PUT (NEWITEM.NUM_PATHS); NEW_LINE;
if NEWITEM.NUM_PATHS < 0 then
   PUT ("The number of paths must be at least zero."); NEW_LINE;
   raise PROGRAM_ERROR;
end if;
if NEWITEM.NUM_PATHS = 0 then
   TEMP := new DISTRIBUTION_RECORD;
   TEMP.WHERE_TO_GO := 0;
   TEMP.PROBABILITY := 100;
   DIST_INSERT (TEMP, NEWITEM.DISTRIBUTION);
else
   TOTAL_PROBABILITY := 101;
   while TOTAL_PROBABILITY > 100 loop
      TOTAL_PROBABILITY := 0;
      for COUNTER in 1 .. NEWITEM.NUM_PATHS  loop
          TEMP := new DISTRIBUTION_RECORD;
```

```
            TEMP.WHERE_TO_GO := INTGET (INPUT_FILE);
            if TEMP.WHERE_TO_GO > SYS.HIGHEST_SERVER then
               SYS.HIGHEST_SERVER := TEMP.WHERE_TO_GO;
            end if;
            TEMP.PROBABILITY := 101;

            while TEMP.PROBABILITY > 100 or TEMP.PROBABILITY < 0 loop
               TEMP.PROBABILITY := INTGET (INPUT_FILE);
--               PUT ("probability = "); PUT (TEMP.PROBABILITY); NEW_LINE;
            end loop;
         DIST_INSERT (TEMP, NEWITEM.DISTRIBUTION);
          TOTAL_PROBABILITY := TOTAL_PROBABILITY + TEMP.PROBABILITY;
         end loop;

      if TOTAL_PROBABILITY > 100 then
           PUT ("The sum of all the probabilities for a single server");
           NEW_LINE;
           PUT ("Should not exceed 100. I'm sorry, but the simulation will");
           NEW_LINE;
           PUT ("have to be aborted."); NEW_LINE;
           raise PROGRAM_ERROR;
      end if;
    end loop; -- of while loop.
 end if; -- of if NUM_PATHS = 0.

 NEWITEM.WHAT_KIND := WHAT_KIND;
 NEWITEM.ASTLOW := LOCAL_ASTLOW;
 NEWITEM.ASTHIGH := LOCAL_ASTHIGH;
 NEWITEM.CAPACITY := CAPACITY;

 NEWITEM.NUMBER_OF_ENTRIES := 0;
 NEWITEM.UTILIZATION := 0.0;
 NEWITEM.MAX_CONTENTS := 0;
 NEWITEM.CURRENT_CONTENTS := 0;
 NEWITEM.AVERAGE_CONTENTS := 0.0;
 NEWITEM.TOTAL_CONTENTS := 0;
 NEWITEM.REQUESTORS_WAITING := 0;
 NEWITEM.AVERAGE_TIME_PER_ENTRY := 0.0;
 NEWITEM.LAST_DEPARTURE := FALSE;
 NEWITEM.TOTALAST := 0;

 NEWITEM.WAIT_Q.Q_PTR := null;
 NEWITEM.WAIT_Q.TOTAL_QENTRIES := 0;
 NEWITEM.WAIT_Q.MAX_QCONTENTS := 0;
 NEWITEM.WAIT_Q.CURRENT_QCONTENTS := 0;
 NEWITEM.WAIT_Q.AVERAGE_QCONTENTS := 0.0;
 NEWITEM.WAIT_Q.ZERO_QENTRIES := 0;
 NEWITEM.WAIT_Q.AVERAGE_QTIME := 0.0;
 NEWITEM.WAIT_Q.TOTAL_QTIME := 0;
 NEWITEM.WAIT_Q.TOTAL_QCONTENTS := 0;
 SERV_INSERT (NEWITEM, SERV_LIST, SYS.SERVER_CREATED); -- POSSIBLE BAD MOVE
                                                       -- using SYS.SERVER_CREATED.

 if SYS.SERVER_CREATED then
    CURRENT_SERVER := NEWITEM;
 end if;

 SYS.SERVER_COUNT := SYS.SERVER_COUNT + 1;
 SERVER_COUNT := SYS.SERVER_COUNT;  -- parameter assignments.
 HIGHEST_SERVER := SYS.HIGHEST_SERVER;
 exception
```

```
when others =>
     PUT ("CREATE_SERVER procedure has failed."); NEW_LINE;
     raise PROGRAM_ERROR;
end CREATE_SERVER;

-------------------------------------------------
with TEXT_IO;
with ADA_IO; use ADA_IO;
with SIMPACK; use SIMPACK;

-- Procedure SIMULATE is the main procedure for this program.
-- It opens the input file SIM.DAT, reads the system data out of it,
-- calls GENERATE_ARRIVAL to put an initial arrival event on the queue,
-- calls CREATE_SERVER, which creates servers and reads the information
-- associated with them from the open file. SIMULATE then calls
-- START_SIMULATION, and if it terminates normally calls PRINT_STATS
-- to print the statistics of interest to the user. It then calls
-- RESET to "clean" the data space and terminates.
-- If it receives an exception from any subprogram it prints out a short
-- message and exits.

procedure SIMULATE is
   IATLOW : INTEGER := -10;
   IATHIGH : INTEGER := -10;
   ASTLOW : INTEGER := 18;
   ASTHIGH : INTEGER := 22;
   CAPACITY : INTEGER := 1;
   TIME: INTEGER := -1;
   START: BOOLEAN := FALSE;
   ANSWER: CHARACTER := 'a';
   CONDITION: STOPTYPE := ATIME;
   HIGHEST_SERVER, SERVER_COUNT: INTEGER := 0;
   DIST: PROBABILITY_DISTRIBUTION;
   INPUT_FILE: TEXT_IO.FILE_TYPE;
begin

TEXT_IO.OPEN (INPUT_FILE, TEXT_IO.IN_FILE,"sim.dat");

while ANSWER /= 'T' and ANSWER /= 't' and ANSWER /= 'R' and ANSWER /= 'r' and ANSWER
/= 's' and ANSWER /= 'S' loop
      ANSWER := CHARGET (INPUT_FILE);
end loop;

if ANSWER = 'T' or ANSWER = 't' then
   CONDITION := ATIME;
elsif ANSWER = 'S' or ANSWER = 's' then
   CONDITION := STARVE;
else CONDITION := REQUESTORS;
end if;

TIME := INTGET (INPUT_FILE);
if TIME < 1 then
   PUT ("That time is invalid!"); NEW_LINE;
   PUT ("Simulation aborted."); NEW_LINE;
   raise PROGRAM_ERROR;
end if;

while ANSWER /= 'u' and ANSWER /= 'U' and ANSWER /= 'e' and ANSWER /= 'E' loop
      ANSWER := CHARGET (INPUT_FILE);
end loop;
```

```
if ANSWER = 'e' or ANSWER = 'E' then
   DIST := EXPONENTIAL;
else DIST := UNIFORM;
end if;

ANSWER := 'a';
IATLOW := INTGET (INPUT_FILE);
if IATLOW < 1 then
   PUT ("Arrival times of less than one are not allowed."); NEW_LINE;
   PUT ("Simulation aborted."); NEW_LINE;
   raise PROGRAM_ERROR;
end if;

if DIST = UNIFORM then
   IATHIGH := INTGET (INPUT_FILE);
   PUT ("IATHIGH = "); PUT (IATHIGH); NEW_LINE;
   if IATHIGH < 1 then
      PUT ("Arrival times of less than one are not allowed."); NEW_LINE;
      PUT ("Simulation aborted."); NEW_LINE;
      raise PROGRAM_ERROR;
   end if;
else
IATHIGH := 2 ** 15 -1;
end if;
GENERATE_ARRIVAL (IATLOW, IATHIGH, DIST);

while not START loop
CREATE_SERVER (HIGHEST_SERVER, SERVER_COUNT, INPUT_FILE);
if HIGHEST_SERVER = SERVER_COUNT then
   START := TRUE;
end if;
end loop;

TEXT_IO.CLOSE (INPUT_FILE);

START_SIMULATION (DIST, IATLOW, IATHIGH, CONDITION, TIME);
PRINT_STATS;
RESET;
exception
when TEXT_IO.NAME_ERROR =>
     PUT ("Data file not found."); NEW_LINE;
when others =>
     PUT ("Simulation aborted."); NEW_LINE;
end SIMULATE;
```

Figure 4 - Listing of Procedures that Access Data File SIM.DAT

2.0 DATA STORAGE:

Data from the input file:

System data:

There are 5 pieces of data taken from the input file describing the system: The condition under which we terminate the simulation, the time (or number of requestors) at which termination (or starvation) occurs, the distribution for the RNG for the arrivals, and the range of values (or mean) for the

RNG.

All this data passes through several intermediate variables during the procedure, but in the end it is all stored in the variable SYS, which is of SYSTEM_TYPE. SYS is global to the package, which means it can be accessed by any subprogram except SIMULATE. SYSTEM_TYPE is a record containing almost all the data needed to describe the simulation in the large except server data. There are two other global variables: FINISHED, which is set to TRUE if the simulation should be terminated now, and ERROR, which is used to decide whether to print a certain error message or not.

Server data:

For every server there corresponds a single record of SERVER_TYPE, which contains almost everything connected with the servers, including queues and pointers to other servers.

The data for all the servers is placed in a linked list of SERVER_TYPE. This allows a user to create as many servers as (s)he wishes dynamically. This list is stored in the same place as SYS: global to the package, making it visible to everything but SIMULATE.

Figure 5 shows the specification of the data types and the package's global variables.

```
package SIMPACK is
type PROBABILITY_DISTRIBUTION is (UNIFORM,EXPONENTIAL,NORMAL,DISCRETE);
-- This type tells what distribution the random number generator will
-- use. Note that NORMAL and DISCRETE remain unimplemented at this point.

type INTEGER_32 is range -2 ** 31 .. 2 ** 31 -1;
-- A 32-bit integer type.  Used only for testing to see if MAXINT
-- (2  ** 15 -1) has been exceeded.

type EVENTCLASS is (ARRIVE, DEPART, STOPSIMULATION, SAMPLE);

-- An enumerated type describing the different kinds of events
-- that may occur during the simulation. At present, they include:
--                1. The arrival or departure of a simulation.
--                2. The termination of the simulation.
--                3. The queue sizes and various other things are sampled.

type STOPTYPE is (REQUESTORS, ATIME, STARVE);
-- An enumerated type describing the three different normal exit
-- conditions for the simulation: when a certain number of requestors
-- has left the system, when a certain time is reached on the clock,
-- and starvation, which occurs when after a certain time the system
-- prevents the arrival of any more requestors.  The system goes on
-- to finish everything already on the queues and in system, then
-- terminates.


type SCHEDULENODE;
type SCHEDULELINK is access SCHEDULENODE;
type SCHEDULENODE is record
   CLASS: EVENTCLASS;       -- type of event
   ACTIVATETIME: INTEGER;     -- when event should occur
```

```
   MAX_ARRIVAL_TIME:INTEGER;    -- length of time for event
   MIN_ARRIVAL_TIME: INTEGER; -- tells what the minimum starting
                              -- value is.
   WHERE_AMI : INTEGER;    -- Used to tell which server the current event
                           -- is occurring at.
   FROM_SERVER: BOOLEAN;    -- Tells whether the last event
                           -- was from a server or not (This only
                           -- makes sense if the last event was an arrival).
   NEXT: SCHEDULELINK;      -- pointer for list membership
end record;
-- Describes an event that occurs during the simulation. Since all events
-- are stored on the queue, it has a pointer type associated with it.

type REQUESTNODE;
type REQUESTLINK is access REQUESTNODE;
type REQUESTNODE is record
   WHERE_NEXT: INTEGER;     -- Tells what server to visit next.
   CREATIONTIME: INTEGER;     -- When requestor was created.
   NEXT: REQUESTLINK;       -- Pointer for list membership.
end record;
-- represents a requestor.



type QUEUE_TYPE is record
     Q_PTR: REQUESTLINK;         -- A pointer to the queue proper.
     MAX_QCONTENTS: INTEGER;     -- Maximum contents.
     AVERAGE_QCONTENTS: FLOAT; -- Average contents during simulation.
     TOTAL_QENTRIES: INTEGER;    -- Total entries into the queue.
     ZERO_QENTRIES: INTEGER;     -- Number of entries waiting 0 time in queue.
     AVERAGE_QTIME: FLOAT;       -- Average time in queue during simulation.
     TOTAL_QTIME: INTEGER;         -- Total time in queue.
     TOTAL_QCONTENTS: INTEGER;
     CURRENT_QCONTENTS: INTEGER; -- Contents of queue when simulation stops.
end record;
-- contains the data associated with a queue

type DISTRIBUTION_RECORD;
type DISTRIBUTION_LIST is access DISTRIBUTION_RECORD;
type DISTRIBUTION_RECORD is record
     PROBABILITY: INTEGER; -- Should be in the range 0..100.
     WHERE_TO_GO: INTEGER; -- Tells what server to go to.
     NEXT: DISTRIBUTION_LIST; -- Pointer to list.
end record;

-- Every server has a distribution list associated with it, each
-- node containing the number of a server that can be reached from
-- this server and the probability that it will go there.

type SERVER_TYPE;
type SERVER_LIST_TYPE is access SERVER_TYPE;
type SERVER_TYPE is record
     SERVER_ID: INTEGER; -- Unique server id.
     NUM_PATHS: INTEGER; -- Number of branches to other servers.
     DISTRIBUTION: DISTRIBUTION_LIST; -- A list, each node of which
                                      -- contains the name of one server
                                      -- that can be reached from here,
                                      -- and the probability that it will
                                      -- go there.

     ASTLOW, ASTHIGH: INTEGER;  -- Min/Max service time for this server.
```

```
      UTILIZATION: FLOAT; -- Server utilization.
      AVERAGE_TIME_PER_ENTRY: FLOAT;
      NUMBER_OF_ENTRIES, MAX_CONTENTS, CURRENT_CONTENTS, CAPACITY: INTEGER;
      AVERAGE_CONTENTS: FLOAT;
      TOTAL_CONTENTS : INTEGER;
      REQUESTORS_WAITING: INTEGER; -- number of requestors that are
                                   -- waiting on the queue.  It probably
                                   -- duplicates current contents.

      LAST_DEPARTURE: BOOLEAN; -- Set to TRUE if the last departure from
                               -- this server has been scheduled, FALSE
                               -- otherwise.

      TOTALAST: INTEGER; -- Sum of all the service times for every entry
                         -- in the server. Divide by number of entries
                         -- to obtain the average time per entry.


      NEXT: SERVER_LIST_TYPE; -- Pointer to list.
      WAIT_Q : QUEUE_TYPE; -- Pointer to the queue holding the waiting
                           -- requestors.
      PROCESS_Q: REQUESTLINK; -- A queue to hold the requestors in process.
      WHAT_KIND: PROBABILITY_DISTRIBUTION; -- Tells whether this server
                                           -- has an exponential, uniform,
                                           -- discrete or normal distribution
                                           -- for its service times.
end record;
-- Describes a server.


type SYSTEM_TYPE is record
   SAMPLES: INTEGER; -- The number of samples gathered.
   SERVER_COUNT: INTEGER; -- Number of servers
   CLOCK: INTEGER;  -- Simulation clock
   STOPTIME: INTEGER; -- stop-simulation time
   HIGHEST_SERVER: INTEGER; -- Highest server number.
   IATLOW, IATHIGH: INTEGER;  -- Lowest and highest arrival times.
   REQUESTS_PROCESSED: INTEGER; -- Number of requestors that have passed
                                -- through the system.
   TOTAL_REQUESTORS: INTEGER;   -- Total number of requestors that should
                                -- be processed before the simulation
                                -- terminates.  This field only makes sense
                                -- if the STOP_CONDITION = REQUESTORS.

   ARRIVAL_GENERATED, SERVER_CREATED, SIMULATION_RUN : BOOLEAN;
                        -- Checks to see whether a simulation can be run
                        -- safely. A simulation cannot run if an
                        -- initial arrival event has not been put on the
                        -- event queue.  Thus, ARRIVAL_GENERATED must be
                        -- TRUE.  Similarly, we must have at least one
                        -- server in the system.  Thus, SERVER_CREATED must
                        -- be TRUE.  Finally, the program cannot be
                        -- run if the data structures are still 'dirty' from
                        -- a previous run.  Thus, SIMULATION_RUN must be
                        -- FALSE.  Also, PRINT_STATS cannot execute if
                        -- a simulation has not been run, and so PRINT_STATS
                        -- will not work unless SIMULATION_RUN is TRUE.

   DISTRIBUTION_TYPE: PROBABILITY_DISTRIBUTION;
                        -- Tells what random number function to use
                        -- for the arrival times.
```

```
   CONDITION: STOPTYPE; -- Describes what condition will terminate the
                        -- simulation under normal circumstances.

   ITEM, HEAD, LAST_EVENT: SCHEDULELINK; -- Pointers to the list
                                         -- of events. ITEM is more or
                                         -- less a dummy variable while
                                         -- HEAD points to the beginning
                                         -- of the list and LAST_EVENT
                                         -- points to an event just after
                                         -- it is removed from the head of
                                         -- the list.  It will be used for
                                         -- determining which server the last
                                         -- event took place at.
end record;

-- contains all data used by the simulation system
-- but not associated with either requestors or servers.

.
.
.

package body SIMPACK is
SERV_LIST, CURRENT_SERVER : SERVER_LIST_TYPE;
          -- A list of servers, and a pointer to the current server.
SYS : SYSTEM_TYPE;
          -- A global variable containing information about the
          -- system in general.

ERROR, FINISHED: BOOLEAN := FALSE;
          -- ERROR is used by START_SIMULATION to determine if
          -- the error message "queue time has reached MAXINT"  has been
          -- printed out. FINISHED is set to TRUE when the simulation is
          -- finished, and is FALSE the rest of the time.
```

Figure 5 - Specification of the Data Types and the Package's Global Variables

3.0 SYSTEM ARCHITECTURE:

It is now time to describe the operation of the program. In order to do this I ask you to visualize three object types, if you can: SERVERS, REQUESTORS, and EVENTS. Figure 7 shows the concept of a server.

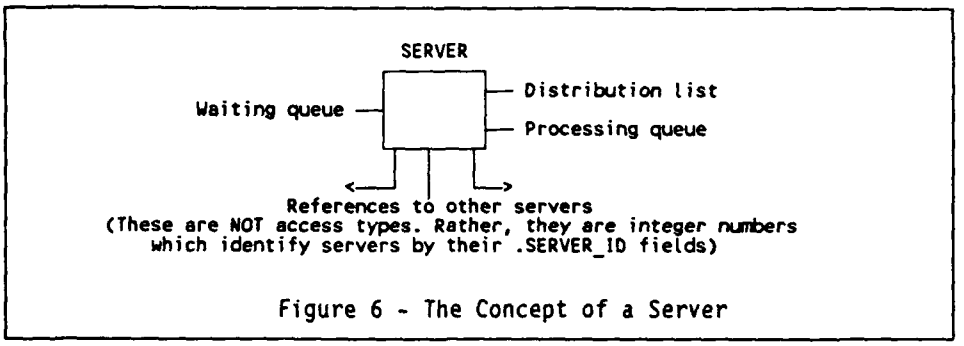


Figure 6 - The Concept of a Server

3.1 SERVERS

Server characteristics:

Distribution of service times
Minimum/Mean distribution time
Maximum distribution (Applicable only to uniform distributions)
Server capacity.
Number of paths to other servers
A pointer to a linked list. Each field of the linked list contains a number identifying a server and a number between 0 and 100 telling how likely it is that a specific requestor will go to that server.

Server Statistics:

Server utilization
Average time per entry.
Number of entries.
Maximum and current server contents.
Average contents.

Other:

Total server contents (used to calculate average contents).
Flag to determine whether the last departure from this server has been scheduled or not (this is determined to occur if the time on the clock + the calculated service time is greater than the simulation time to stop and SYS.CONDITION = ATIME). If this flag is set to true that will mean that the server will be busy until the end of the simulation. This means that the calculation of the total in-use server time (which is part of the calculation of the average time per entry for this server) will have to be modified: Instead of simply adding the service time for this server to it we must add (simulation stop time - current time).

Total service time (the sum of the service times for all requestors that have been serviced here. It is used to calculate the average time per entry).

Server ID (an integer number. Servers are numbered with the first servers created first. Note that the server ordering must be continuous -- if there is a server 6 there must also be servers 1, 2, 3, 4 and 5).

A waiting queue where requestors waiting to be processed by this server will be stored, and a processing queue where requestors being serviced are stored before they leave.

The reason for the process queue is that it is sometimes convenient to hold the data concerning requestors while they are in process. When we want to know what the current contents of the server are we simply count the contents of the queue. Also, in this package we decide where a requestor should go next when they arrive, rather than when they depart. Thus, we need to record the destination (WHERE_NEXT) field between arrival and departure events. So we store it on this process queue so we will know where to send it after the server finishes with it. This is an example of a feature that proved more useful at the time when we first created this program but has since been made almost useless due to the many changes that have been made to this program.

Queue statistics:

Maximum, current and average queue contents.
Total number of queue entries.
Number of zero-time queue entries.
Average time in the queue.

Other: Total time in the queue. (Used to calculate the average time in the queue).

3.2 REQUESTORS

Requestors have two fields and a NEXT field because they are designed to be part of a linked list.

WHERE_NEXT: Tells where a requestor should go after it leaves the server it is now at. This field is an integer value such that 2 means go to server #2, 1 to server #1, etc. A 0 value in this field means that the requestor is to leave the system when it leaves the current server.

Creation time: Records the time at which it was created.

Note that a requestor which is new to the system must always arrive first at server #1.

3.3 EVENTS

Events are stored on the event queue, which is a part of the SYS global variable.

An event is described by six pieces of information:

```
CLASS: EVENTCLASS;        -- type of event, which can be an ARRIVAL,
                             DEPARTURE, SAMPLE, or the simulation termination
                             (STOPSIMULATION).
ACTIVATETIME: INTEGER;    -- when event should occur
MAX_ARRIVAL_TIME:INTEGER;
MIN_ARRIVAL_TIME: INTEGER;
WHERE_AMI : INTEGER;      -- Used to tell which server the current event
                          -- is occurring at.
FROM_SERVER: BOOLEAN;     -- Tells whether the last event
                          -- was from a server or not (This only
                          -- makes sense if the last event was an arrival).
```

There is also a NEXT field, since an EVENT is meant to be stored on a linked list.

MAX_ and MIN_ARRIVAL_TIME is used only when we are scheduling the initial arrival event. They appear to have outlived their usefulness and should probably be removed.

3.4 OPERATIONS ON DATA OBJECTS

SERVERS, REQUESTORS and EVENTS are merely static data objects that are modified and moved around by the program. The procedure that decides how these objects are handled is START_SIMULATION.

3.4.1 START_SIMULATION

Requirements for START_SIMULATION:
1. The necessary data structures must exist and they must be "clean."
2. An initial arrival event must be on the event queue.
3. At least one server must be on the server list.
4. Time for simulation must be greater than zero, if the ending condition is either STARVE or TIME. If it isn't the number of requestors that must leave the system before termination must be greater than zero.
5. There must be the RIGHT number of servers. An exception will be generated if, for instance, a user creates pointers to seven servers but only generates four.

Results of START_SIMULATION:
A simulation has now been run to completion, and a bunch of statistics are now stored with each server, with the queue associated with each server, and in the system global variable. They are printed out by PRINT_STATS.

START_SIMULATION performs the following loop until the simulation terminates:

```
while not FINISHED loop
    if SYS.HEAD /= null and SYS.CLOCK < SYS.HEAD.ACTIVATETIME
    then
       SYS.CLOCK := SYS.HEAD.ACTIVATETIME;
    end if;
    if SYS.REQUESTS_PROCESSED >= SYS.TOTAL_REQUESTORS-1
    and SYS.CONDITION = REQUESTORS then
       SYS.STOPTIME := SYS.LAST_EVENT.ACTIVATETIME; -- May be wrong.
       FINISHED := TRUE;
    elsif SYS.HEAD.CLASS = ARRIVE then
       SYS.LAST_EVENT := SYS.HEAD;
       SYS.HEAD := SYS.HEAD.NEXT;          -- remove top node
       HANDLE_ARRIVAL;
    elsif SYS.HEAD.CLASS = DEPART then
       SYS.LAST_EVENT := SYS.HEAD;
       SYS.HEAD := SYS.HEAD.NEXT;
       HANDLE_DEPARTURE;
    elsif SYS.HEAD.CLASS = STOPSIMULATION then
          SYS.STOPTIME := SYS.HEAD.ACTIVATETIME;
          SYS.HEAD := SYS.HEAD.NEXT;
          FINISHED := TRUE;
    elsif SYS.HEAD.CLASS = SAMPLE then
          SYS.HEAD := SYS.HEAD.NEXT;
          if SYS.CLOCK > SYS.STOPTIME and SYS.CONDITION = STARVE then
             FINISHED := TRUE;
             SYS.ITEM := SYS.HEAD;
             -- This next loop determines whether the system has starved
             -- or not.  What we do is check every item on the event queue
             -- and if there are no arrivals or departures on the queue
             -- then the system has starved and the simulation should
             -- terminate.
             while SYS.ITEM /= null loop
                   if SYS.ITEM.CLASS = ARRIVE or SYS.ITEM.CLASS = DEPART then
                      FINISHED := FALSE;
                   end if;
                   SYS.ITEM := SYS.ITEM.NEXT;
              end loop;
          end if;
          if not FINISHED then
          for COUNTER in 1..SYS.SERVER_COUNT loop
          CURRENT_SERVER := SERV_FINDITH (SERV_LIST, COUNTER);
          CURRENT_SERVER.WAIT_Q.TOTAL_QCONTENTS :=
             CURRENT_SERVER.WAIT_Q.TOTAL_QCONTENTS +
             CURRENT_SERVER.WAIT_Q.CURRENT_QCONTENTS;
          CURRENT_SERVER.TOTAL_CONTENTS := CURRENT_SERVER.TOTAL_CONTENTS +
             CURRENT_SERVER.CURRENT_CONTENTS;
          end loop;

          SYS.SAMPLES := SYS.SAMPLES + 1;
          SYS.ITEM := new
              SCHEDULENODE'(SAMPLE,SYS.CLOCK+SINTERVAL,0,0,0,FALSE,null);
          EVENT_INSERT (SYS.ITEM, SYS.HEAD);

       end if; -- if we are not finished in sample.
    end if;  -- of the outermost if.
end loop;
```

Figure 7 - STAT_SIMULATION Loop

Someone is bound to ask "what is the rationale for the SAMPLE event ?" Well, some of the statistics that we collate, such as average server contents, are measures over time. It seemed silly to check every server every time unit to see what contents they held, and them up and divide by the time at the end to get the average. Instead, we check them at fixed time intervals, and after the simulation is over we divide by the number of samples to get the average for these statistics. The statistics that must be checked this way are the average contents of a server and the average contents of a queue associated with a server.

Following is Figure 8, which contains the subprograms called by the loop in Figure 7.

```
--
-- This procedure handles an arrival when it occurs. The first thing
-- it does is find out which server the arrival event is occurring at and
-- make it the current server.  Next, it checks to see whether the arrival
-- event has just occurred at server 1. If it has, and it has just been
-- generated as opposed to being directed there from another server, the
-- next arrival at server one from outside the system is scheduled.
-- Statistics are then updated.  If the current server can process
-- the requestor, SCHEDULE_DEPARTURE is called.
-- In case of an exception, PROGRAM_ERROR is propagated to the calling unit.
procedure HANDLE_ARRIVAL is
   REQUESTOR: REQUESTLINK := null;
   NUM_ARRIVE: INTEGER := 0;
   IAT: INTEGER := 0;
begin
   CURRENT_SERVER := SERV_FINDITH (SERV_LIST,SYS.LAST_EVENT.WHERE_AMI);


   if SYS.LAST_EVENT.WHERE_AMI = 1 and not SYS.LAST_EVENT.FROM_SERVER then
      if not (SYS.CONDITION = STARVE and SYS.CLOCK > SYS.STOPTIME) then

         IAT := GENERATE (SYS.IATLOW, SYS.IATHIGH, SYS.DISTRIBUTION_TYPE);
         SCHEDULE_ARRIVAL (1, IAT, FALSE);
      else PUT ("Aborted new scheduled departure."); NEW_LINE;
      end if;
   end if;

      REQUESTOR := new REQUESTNODE'(0,SYS.CLOCK,null);
      REQUESTOR.WHERE_NEXT := SET_DESTINATION (REQUESTOR, CURRENT_SERVER);
      CURRENT_SERVER := SERV_FINDITH (SERV_LIST, SYS.LAST_EVENT.WHERE_AMI);
      INQUEUE (REQUESTOR, CURRENT_SERVER.WAIT_Q.Q_PTR);

      CURRENT_SERVER.WAIT_Q.TOTAL_QENTRIES := CURRENT_SERVER.WAIT_Q.TOTAL_QENTRIES + 1;
      CURRENT_SERVER.WAIT_Q.CURRENT_QCONTENTS := QCONTENTS (CURRENT_SERVER.WAIT_Q.Q_PTR);

      if CURRENT_SERVER.WAIT_Q.CURRENT_QCONTENTS > CURRENT_SERVER.WAIT_Q.MAX_QCONTENTS then
        CURRENT_SERVER.WAIT_Q.MAX_QCONTENTS := CURRENT_SERVER.WAIT_Q.CURRENT_QCONTENTS;
      end if;

     if (CURRENT_SERVER.CURRENT_CONTENTS < CURRENT_SERVER.CAPACITY) then
         SCHEDULE_DEPARTURE;
         REQUESTOR := new REQUESTNODE;
           REQUESTOR.WHERE_NEXT := CURRENT_SERVER.WAIT_Q.Q_PTR.WHERE_NEXT;
           REQUESTOR.CREATIONTIME := CURRENT_SERVER.WAIT_Q.Q_PTR.CREATIONTIME;
           REQUESTOR.NEXT := null;
         INQUEUE (REQUESTOR, CURRENT_SERVER.PROCESS_Q);
         CURRENT_SERVER.WAIT_Q.Q_PTR := CURRENT_SERVER.WAIT_Q.Q_PTR.NEXT;
         CURRENT_SERVER.CURRENT_CONTENTS := QCONTENTS (CURRENT_SERVER.PROCESS_Q);

         if CURRENT_SERVER.CURRENT_CONTENTS > CURRENT_SERVER.MAX_CONTENTS then
            CURRENT_SERVER.MAX_CONTENTS := CURRENT_SERVER.CURRENT_CONTENTS;
         end if;

         CURRENT_SERVER.WAIT_Q.CURRENT_QCONTENTS := QCONTENTS (CURRENT_SERVER.WAIT_Q.Q_PTR);
         -- remove from queue.

     else  -- if the server is busy.
          CURRENT_SERVER.REQUESTORS_WAITING := QCONTENTS (CURRENT_SERVER.WAIT_Q.Q_PTR);

     end if; -- if server not busy
exception
when others =>
     PUT ("HANDLE_ARRIVAL has failed."); NEW_LINE;
     raise PROGRAM_ERROR;
end HANDLE_ARRIVAL;
```

```
-------------------------------------------------

--
-- This procedure handles the departure of requestors
-- from a server.  The first thing it does is to find out where
-- the departure event is supposed to occur, and make that the current
-- server.  Having done this, it kicks the first requestor on the "processing"
-- queue off and updates all statistics associated with this event.  Next,
-- if there are any requestors on the queue waiting to be processed it calls
-- SCHEDULE_DEPARTURE to pull one off the waiting queue and start processing
-- it.  In the event of an exception PROGRAM_ERROR will be propagated to the
-- calling unit.
--

procedure HANDLE_DEPARTURE is
   REQUESTOR: REQUESTLINK := null;
   NUM_ARRIVE: INTEGER := 0;
   DUMMY: INTEGER; -- UNNECESSARY IN FINAL PRODUCT.
   HUGE1, HUGE2, HUGE3: INTEGER_32;
begin

   CURRENT_SERVER := SERV_FINDITH (SERV_LIST, SYS.LAST_EVENT.WHERE_AMI);
   DUMMY := QCONTENTS (CURRENT_SERVER.WAIT_Q.Q_PTR);

   if CURRENT_SERVER.PROCESS_Q.WHERE_NEXT /= 0 then
     SCHEDULE_ARRIVAL (CURRENT_SERVER.PROCESS_Q.WHERE_NEXT,0, TRUE);
   else
      SYS.REQUESTS_PROCESSED := SYS.REQUESTS_PROCESSED + 1;
   end if;



  CURRENT_SERVER.PROCESS_Q := CURRENT_SERVER.PROCESS_Q.NEXT;
  CURRENT_SERVER.CURRENT_CONTENTS := QCONTENTS (CURRENT_SERVER.PROCESS_Q);

-- This probably won't happen here, because we are removing, and a shrinkage
-- will not exceed the maximum. RECOMMEND REMOVAL.

  if CURRENT_SERVER.CURRENT_CONTENTS > CURRENT_SERVER.MAX_CONTENTS then
     CURRENT_SERVER.MAX_CONTENTS := CURRENT_SERVER.CURRENT_CONTENTS;
  end if;

-- END REMOVAL.

if CURRENT_SERVER.REQUESTORS_WAITING > 0 then
         SCHEDULE_DEPARTURE;
         REQUESTOR := new REQUESTNODE;
           REQUESTOR.WHERE_NEXT := CURRENT_SERVER.WAIT_Q.Q_PTR.WHERE_NEXT;
           REQUESTOR.CREATIONTIME := CURRENT_SERVER.WAIT_Q.Q_PTR.CREATIONTIME;
           REQUESTOR.NEXT := null;
         INQUEUE (REQUESTOR, CURRENT_SERVER.PROCESS_Q);
         CURRENT_SERVER.CURRENT_CONTENTS := QCONTENTS (CURRENT_SERVER.PROCESS_Q);
         if CURRENT_SERVER.CURRENT_CONTENTS > CURRENT_SERVER.MAX_CONTENTS then
            CURRENT_SERVER.MAX_CONTENTS := CURRENT_SERVER.CURRENT_CONTENTS;
         end if;
         CURRENT_SERVER.WAIT_Q.Q_PTR := CURRENT_SERVER.WAIT_Q.Q_PTR.NEXT;
         CURRENT_SERVER.WAIT_Q.CURRENT_QCONTENTS := QCONTENTS (CURRENT_SERVER.WAIT_Q.Q_PTR);
         -- remove from queue.

         CURRENT_SERVER.REQUESTORS_WAITING := QCONTENTS (CURRENT_SERVER.WAIT_Q.Q_PTR);
end if;

exception
when others =>
     PUT ("HANDLE_DEPARTURE has failed. "); NEW_LINE;
     raise PROGRAM_ERROR;
end HANDLE_DEPARTURE;
```

Figure 8 - Listing of Subprograms Called by the Loop in START_SIMULATION

Following is Figure 9, which contains listings of the subprograms called by START_SIMULATION:

```
--
-- Almost the first thing SCHEDULE_DEPARTURE does is check to see
-- if it CAN schedule a departure -- if the service time + the current
-- time is greater than the time when the simulation stops ( if time is the
-- stop condition for the simulation), it should not schedule a departure.
-- If the program finds that it cannot schedule a departure, it sets a flag
-- on the current server called LAST_DEPARTURE, telling itself that the
-- server will be busy with an object until the simulation terminates.
-- On the other hand, if the event passes the above test,
-- SCHEDULE_DEPARTURE schedules a departure event for the first
-- requestor on the current server's waiting queue. Having done this,
-- it moves the requestor from the current server's waiting queue to the
-- current server's processing queue, where requestors that are being processed
-- are stored. It then updates statistics both for the current server and for
-- the current server's queue. If an exception occurs PROGRAM_ERROR is
-- propagated to the calling unit.
--
-- The procedure also checks to see if total queue time has reached MAXINT
-- (32767) if it does, than the simulation is prematurely terminated.
-- This is the only value checked here for this occurrence because
-- this field grew much faster than the other fields during testing.
--

procedure SCHEDULE_DEPARTURE is
AST: INTEGER := 0;
BIG1, BIG2, BIG3: INTEGER_32;
begin
     AST := GENERATE (CURRENT_SERVER.ASTLOW, CURRENT_SERVER.ASTHIGH, CURRENT_SERVER.WHAT_KIND);


if (SYS.CLOCK+AST < SYS.STOPTIME) or (SYS.CONDITION /= ATIME) then
    SYS.ITEM := new SCHEDULENODE;
    SYS.ITEM.CLASS := DEPART;
    SYS.ITEM.ACTIVATETIME := AST + SYS.CLOCK;
    SYS.ITEM.MAX_ARRIVAL_TIME := AST;
    SYS.ITEM.MIN_ARRIVAL_TIME := 0;
    SYS.ITEM.WHERE_AMI := SYS.LAST_EVENT.WHERE_AMI;
    SYS.ITEM.FROM_SERVER := TRUE;
    SYS.ITEM.NEXT := null;

    CURRENT_SERVER.TOTALAST := CURRENT_SERVER.TOTALAST + AST;
    EVENT_INSERT (SYS.ITEM, SYS.HEAD);

elsif not CURRENT_SERVER.LAST_DEPARTURE then
      CURRENT_SERVER.LAST_DEPARTURE := TRUE;
        CURRENT_SERVER.TOTALAST := CURRENT_SERVER.TOTALAST +
                                   (SYS.STOPTIME - SYS.CLOCK);
end if;

BIG1 := INTEGER_32 (CURRENT_SERVER.WAIT_Q.TOTAL_QTIME);
BIG2 := INTEGER_32 (SYS.CLOCK);
BIG3 := INTEGER_32 (CURRENT_SERVER.WAIT_Q.Q_PTR.CREATIONTIME);
if BIG1 + (BIG2- BIG3) < 2 ** 15 - 1 then
   CURRENT_SERVER.WAIT_Q.TOTAL_QTIME := CURRENT_SERVER.WAIT_Q.TOTAL_QTIME +
                (SYS.CLOCK - CURRENT_SERVER.WAIT_Q.Q_PTR.CREATIONTIME);
else
   CURRENT_SERVER.WAIT_Q.TOTAL_QTIME := 2 ** 15 - 1;
   FINISHED := TRUE;
   if not ERROR then
   PUT ("The total queue time has just gotten as large as MAXINT."); NEW_LINE;
   PUT ("The program will be terminated prematurely.");
   ERROR := TRUE;
   end if;
end if;
if (SYS.CLOCK - CURRENT_SERVER.WAIT_Q.Q_PTR.CREATIONTIME) = 0 then
   CURRENT_SERVER.WAIT_Q.ZERO_QENTRIES := CURRENT_SERVER.WAIT_Q.ZERO_QENTRIES + 1;
end if;


CURRENT_SERVER.CURRENT_CONTENTS := QCONTENTS (CURRENT_SERVER.PROCESS_Q);
CURRENT_SERVER.NUMBER_OF_ENTRIES := CURRENT_SERVER.NUMBER_OF_ENTRIES + 1;
if CURRENT_SERVER.CURRENT_CONTENTS > CURRENT_SERVER.MAX_CONTENTS then
   CURRENT_SERVER.MAX_CONTENTS := CURRENT_SERVER.CURRENT_CONTENTS;
end if;
exception
when others =>
PUT ("SCHEDULE_DEPARTURE has failed. "); NEW_LINE;
raise PROGRAM_ERROR;
end SCHEDULE_DEPARTURE;
```

```
-------------------------------------------------
--
-- An incredibly trivial procedure. It schedules an arrival
-- at server INDEX, to occur after ARRIVAL time units have
-- elapsed after the current time. If the requestor is departing
-- from a server, FROM_SERV is TRUE, else FALSE.
--
procedure SCHEDULE_ARRIVAL (INDEX: in INTEGER; ARRIVAL: in INTEGER; FROM_SERV: BOOLEAN) is
begin

      SYS.ITEM := new SCHEDULENODE'(ARRIVE,ARRIVAL+SYS.CLOCK,0,0,INDEX,FROM_SERV,null);

      EVENT_INSERT (SYS.ITEM, SYS.HEAD);
exception
when others =>
     PUT ("SCHEDULE_ARRIVAL has failed. "); NEW_LINE;
end SCHEDULE_ARRIVAL;
```

Figure 9 - Listings of Subprograms Called by the Loop in START_SIMULATION

The various insertion procedures are not listed here, because they are simple linked list functions. The same goes for SERV_FINDITH.

4.0 OUTPUT

All output is sent to standard output, and if all goes well then the only procedure that will print out anything is PRINT_STATS, which prints out the statistics we have deemed to be of interest to the user. Of course, most of the subprograms do print out some kind of message if an error occurs in them, but with any luck the only message you will see will be from PRINT_STATS. Figure 10 contains the code for PRINT_STATS.

```
-- This procedure prints out the statistics of interest to a user.
-- it may only be used after a simulation is run.
procedure PRINT_STATS is
NO_SIMULATION : EXCEPTION;
begin
if not SYS.SIMULATION_RUN then
   raise NO_SIMULATION;
end if;

PUT ("Simulation Stopped at Time:  "); PUT (SYS.CLOCK);
NEW_LINE;
CURRENT_SERVER := SERV_LIST;
PUT (SYS.SERVER_COUNT); PUT (" servers in this run."); NEW_LINE;
PUT ("SERVER   CAPACITY  ENTRIES  AVG. TIME   MAX    CURRENT   AVG.       UTIL. ");
NEW_LINE;
PUT ("                           PER  ENTRY     \     |   /           ");
NEW_LINE;
PUT ("                                          CONTENTS             ");
NEW_LINE;
for COUNT in 1..SYS.SERVER_COUNT loop
    PUT (CURRENT_SERVER.SERVER_ID,2);
    PUT (CURRENT_SERVER.CAPACITY,12);
    PUT (CURRENT_SERVER.NUMBER_OF_ENTRIES,9);
    PUT (CURRENT_SERVER.AVERAGE_TIME_PER_ENTRY,10);
    PUT (CURRENT_SERVER.MAX_CONTENTS,8);
    PUT (CURRENT_SERVER.CURRENT_CONTENTS,7);
    PUT (CURRENT_SERVER.AVERAGE_CONTENTS,9);
    PUT (CURRENT_SERVER.UTILIZATION*100.0,11); PUT (" %"); NEW_LINE;
    CURRENT_SERVER := CURRENT_SERVER.NEXT;
end loop;

CURRENT_SERVER := SERV_LIST;
NEW_LINE; NEW_LINE; NEW_LINE; NEW_LINE;
PUT ("QUEUE   ENTRIES   0-ENTRIES   AVG. TIME      MAX.   CURRENT     AVG.");
NEW_LINE;
PUT ("                                              CONTENTS         ");
NEW_LINE; NEW_LINE;
for COUNT in 1..SYS.SERVER_COUNT loop
PUT (CURRENT_SERVER.SERVER_ID,2);
PUT (CURRENT_SERVER.WAIT_Q.TOTAL_QENTRIES,12);
PUT (CURRENT_SERVER.WAIT_Q.ZERO_QENTRIES,12);
PUT (CURRENT_SERVER.WAIT_Q.AVERAGE_QTIME,9);
PUT (CURRENT_SERVER.WAIT_Q.MAX_QCONTENTS,10);
PUT (CURRENT_SERVER.WAIT_Q.CURRENT_QCONTENTS,9);
PUT (CURRENT_SERVER.WAIT_Q.AVERAGE_QCONTENTS,9);
NEW_LINE;
CURRENT_SERVER := CURRENT_SERVER.NEXT;
end loop;
NEW_LINE; NEW_LINE; NEW_LINE; NEW_LINE;

exception
when NO_SIMULATION =>
     PUT ("You must run a simulation before I can print out any statistics.");
     NEW_LINE;
     PUT ("PRINT_STATS procedure failed."); NEW_LINE;
when others =>
     PUT ("PRINT_STATS procedure failed."); NEW_LINE;
end PRINT_STATS;
```

Figure 10 - Listing for Procedure PRINT_STATS

5.0 SIMULATION TERMINATION:

Normal Simulation termination occurs when:

1. A specified time has elapsed. This condition is set by the user in the input file, and should only occur if the user explicitly sets the condition to ATIME in the input file.

2. A specified number of requestors have left the system. Again, this condition is explicitly set by the user and should only occur if it is told to.

3. The system starves. By this I mean that it processes all the requestors in the system and no more requestors are arriving.

Abnormal simulation termination occurs when:

Total queue time reaches MAXINT (32767). When this happens the simulation is stopped and all the statistics to date are printed out.

The simulation is aborted when:

1. The user gives funny input (numbers where there should be letters, that sort of thing).
2. There isn't an input file.
3. The user forgets to describe a server in the input file.
4. There aren't enough servers. (This can happen because all servers are strictly ordered 1 .. whatever. If there is a server # 5 then there must also be servers 1, 2, 3, and 4. If a user sets a pointer to server # 6 but only creates three servers, then the simulation would abort).
5. Faulty input data (as opposed to just "funny" input. The difference here is that instead of just giving spurious characters, the user is giving the program correct but absurd data, such as a time to run of 0, or a capacity of 0).

APPENDIX A: STATISTICS GATHERED

For the system:

1. Simulation ending time.
   The user can directly specify an ending time or starvation time, and it is recorded here. If the user specifies that a certain number of requestors should pass through the system instead that is recorded in SYS.TOTAL_REQUESTORS.

   In either case, the last time on the clock is placed in SYS.STOPTIME after the simulation terminates. This occurs in START_SIMULATION.

2. Number of servers used:
   This is stored in SYS.SERVER_COUNT. It is incremented every time a new server is created by CREATE_SERVER.

Server statistics:

1. The server ID. This is assigned by CREATE_SERVER based on when it was created. The first server created is number one, the second is number 2, and so on. Note that a new requestor must always arrive at server number 1, and that there can only be one server on the same level as server #1.

2. Server capacity. This is set by the CREATE_SERVER procedure, and is read in from the input file. A capacity must always be greater than zero.

3. Number of entries on a server. This is initialized to 0 by CREATE_SERVER, and after that it is incremented whenever a requestor moves off a server's waiting queue and onto its process queue. Since SCHEDULE_DEPARTURE is called only when this happens, the update is placed in that procedure.

4. Average time per entry. This is initialized to zero by CREATE_SERVER, and is found after the end of the simulation by dividing the sum of all the service times at this server by the number of entries.

   The sum of all the service times is initialized to 0 by CREATE_SERVER. Every time a server accepts a requestor for processing (this occurs in SCHEDULE_DEPARTURE when a requestor is moved from the waiting queue to the processing queue) we add the time it will take to serve this requestor to the sum. If the simulation will end before the requestor can be completely serviced we add the sum (termination time - current time).

5. Current contents of a server. This is a counter of how many things are being processed by the server, and is obtained by counting the number of things on the server's processing queue using the QCONTENTS function. The value is initialized to 0 and is updated
   A. In the SCHEDULE_DEPARTURE procedure.
   B. In HANDLE_ARRIVAL, when a requestor is taken off a waiting queue and placed in processing.
   C. Twice in HANDLE_DEPARTURE. Once when we take something off the server (when a requestor departs), and a second time if there are requestors waiting and they are immediately put into process.

6. Maximum contents of a server. This is initialized to 0, and afterwards every time the current contents field is updated we compare the new value to the MAX_CONTENTS field. If the value of CURRENT_CONTENTS is greater than the value in MAX_CONTENTS then we set MAX_CONTENTS equal to CURRENT_CONTENTS.

7. Average contents of a server. Initialized to zero, we sample the value of current contents every so often. The average contents is equal to the sum of the sampled values divided by the number of samples. To learn more about samples see the relevant section on SAMPLING.

8. Server utilization. This is the average contents of the server divided by its capacity.

All of the above is part of SERVER_TYPE, which is a record describing a single server. In addition, there is also a QUEUE_TYPE record that is part of SERVER_TYPE, which describes a queue attached to that server.

Most of the statistics are initialized to zero in the procedure RESET or as part of CREATE_SERVER.

Queue Statistics:

1. Queue ID. Since every queue is part of a server's record, a queue ID is simply the ID of the server it is attached to.

2. Number of entries on the queue. Recorded in the TOTAL_QENTRIES field, it is initialized to zero and is incremented every time a new requestor arrives at a queue. This is an arrival event and is handled by the HANDLE_ARRIVAL procedure.

3. Zero Entries. These are the entries on the queue that left the queue the instant they arrived. This is determined to have occurred if, when a requestor is pulled of the waiting queue, the value of it's creation time minus the value of the system clock is zero. This value is initialized to zero and it is changed only in SCHEDULE_DEPARTURE, which is always called when a requestor is moved off the waiting queue.

4. Average time on the queue. This is calculated after the simulation by the sum of all the times on the queue (stored in TOTAL_QTIME) divided by the number of entries. TOTAL_QTIME is initialized to zero, and after that is increased every time schedule departure is called by
   TOTAL_QTIME := TOTAL_QTIME + (SYS.CLOCK - the first thing on the queue's creation time);

   At the end of the simulation we also add up the wait times for the things that are still on the queue.

   Note that TOTAL_QTIME grows very quickly, in all our testing runs much more quickly than anything else. Thus, all our checks to see if a value has grown too large are checked against this value.

5. Current contents of the queue. We get this by counting all the objects on the queue using the QCONTENTS function. We make this change every time we change the length of the queue. That is, whenever an arrival or departure occurs so look for this statistic to be changed in HANDLE_ARRIVAL and HANDLE_DEPARTURE.

6. Maximum contents of the queue. This statistic is almost the same as its counterpart in the server statistics. As before, MAX_QCONTENTS is based on and update of CURRENT_QCONTENTS, and if CURRENT_QCONTENTS is larger than MAX_QCONTENTS := CURRENT_QCONTENTS. The difference is that MAX_CONTENTS for a server is checked whenever a server's CURRENT_CONTENTS are changed,

period. However, the handling for queues is a little smarter in that we know not to check something when something is being pulled off the queue, since when a subtraction occurs it cannot increase above the maximum.

7. Average queue contents. This is checked the same way as server average contents: Every so often we sample the value of queue's current contents, and after the simulation terminates the average queue contents is equal to the sum of the sampled values divided by the number of samples. Again, see SAMPLING for more details on sampling.

All of the queue statistics are part of QUEUE_TYPE, and most are initialized to zero by CREATE_SERVER (a queue and its statistics are initialized at the same time as the server to which it is attached).

APPENDIX B: OTHER ITEMS OF INTEREST

GENERATION OF RANDOM NUMBERS

All requests for a random number call procedure GENERATE, passing variables LOW, HIGH, and WHAT_KIND. LOW and HIGH specify the range LOW..HIGH in which the random number may be generated. WHAT_KIND specifies whether the random number should be distributed uniformly or exponentially. In the event that WHAT_KIND is EXPONENTIAL, LOW will contain the mean value and HIGH will be ignored. We then call either UNIFORM_FUNC or EXP subprograms (which are in RANDOM.ADA) to generate an appropriate random number.

SAMPLING

Sampling is one of the four principle events that may occur during the system. When a sample occurs, the current values of the queue and the server are added to the running totals TOTAL_CONTENTS and TOTAL_QCONTENTS respectively.

How often does a sample occur? If we know that there is a time limit on the simulation, we use UNIFORM to generate a random interval in the range (simulation time / 20) .. (simulation time /10). If the ending condition is based on the number of requestors, we select a random interval in the range

(Minimum arrival time * total requestors /20) ..
(Minimum arrival time * total requestors /10).

Thus, there should usually be between 10 and 20 samples in a simulation, although its anybody's guess how many there will be if the ending condition is REQUESTORS. Hopefully, between 10 and 20 is enough to provide a degree of precision without having to do too much work and slowing the computer down.

The random interval generated is stored in SINTERVAL, and the first sample is scheduled to occur at SINTERVAL. After that whenever a sample event is handled the next sample is scheduled to occur at SYS.CLOCK + SINTERVAL. SINTERVAL is a local variable in START_SIMULATION.

THE SIMULATION CLOCK

The simulation clock is not directly based on the hardware clock. Instead, it is an integer value initialized to zero at procedure RESET. After that, whenever we take an event off the event queue we set clock equal to the event's activate time. This occurs during the main loop in procedure START_SIMULATION mentioned above.

MAXINT

MAXINT is defined to be 2 ** 15 - 1, or 32767. I don't remember when we made the decision to define it to be this on a 32-bit machine, but there it is. However, since it IS a 32 bit machine, we are able to check whether MAXINT has been exceeded very easily. Before we do any addition that we think will exceed the value, we convert the values to a 32-bit number and add. If the 32-bit number is greater than MAXINT, then we prematurely terminate the exception, setting the relevant value to MAXINT. During our testing TOTAL_QTIME, which is one of QUEUE_TYPE's fields, was the only one to give us this trouble, and consequently is the only statistic addition guarded this way. Of course, we also have to watch user input and so we have to guard the INTGET function too. You can find these checks in subprograms GETINT, SCHEDULE_DEPARTURE and START_SIMULATION.

Other items of interest about the system:
A new requestor in the system must always arrive at server number one. There can only be one level-1 server. After that, you can redirect as you please.

APPENDIX C: KNOWN BUGS

The TOTAL_QCONTENTS field of the QUEUE_TYPE records seems to grow a trifle quickly. Aside from that, there are no KNOWN bugs.

However, there is one aspect of the program that will probably cause a lot of mistakes and this is in the input file. When we describe a uniform-distribution random number generator we use three pieces of data: the type of RNG, the minimum value and the maximum value. For instance "u 5 10" tells us that this is a uniformly-distributed RNG with a minimum possible value of 5 and a maximum of 10.

However, exponentially-distributed RNGs require only TWO pieces of data: the type of RNG and a mean. For instance, "e 20" tells us that this is an exponentially-distributed RNG with a mean of 20.

It is very easy to slip up and say "e 20 40" or "u 30" by mistake! Especially since there are a **LOT** of other numbers on that line, and it is easy to put in an extra number (or one too few) by mistake.

APPENDIX D: UPGRADES

1. Convert the program so that it entirely uses either 16-bit or 32-bit operations. A program that operated entirely with 16-bit numbers would be compatible with machines as old as the 80286, while an entirely 32-bit program would greatly extend the range of maximum values. The current mixture of 16- and 32- bit numbers is neither compatible with older machinery nor is MAXINT very large. (At present, 16-bit numbers are used everywhere except in calculation of MAXINT, which uses 32-bit numbers.

2. Supply a driver program which would make the creation of a user program much easier.

3. Make the program error messages more friendly. At this time they are very terse and do not make much sense to someone who is not familiar with the source code.

4. Add more types of random number distributions.

5. Allow the program to perform more than one simulation before terminating.

6. Remove REQUESTORS_WAITING field from SERVER_TYPE data type.

7. Eliminate MAX_ and MIN_ARRIVAL_TIME fields from SCHEDULENODE data type.

8. Have the program give some kind of output during processing, to give the user some idea of how long the program has been running and how long the user has to wait before the program terminates.

9. Allow the generation of requestors that take up more than one unit of a server's capacity. At present, all requestors take up exactly one unit of a requestor's capacity.

10. Give the simulation the ability to have more than one entry point into a simulated system. For example, requestors that have just entered the system could enter at server #5 as well as server #1.

11. Make the simulation execute FASTER.

Program Listings

```
with TEXT_IO;
with ADA_IO; use ADA_IO;
with SIMPACK; use SIMPACK;

-- Procedure SIMULATE is the main procedure for this program.
-- It opens the input file SIM.DAT, reads the system data out of it,
-- calls GENERATE_ARRIVAL to put an initial arrival event on the queue,
-- calls CREATE_SERVER, which creates servers and reads the information
-- associated with them from the open file. SIMULATE then calls
-- START_SIMULATION, and if it terminates normally calls PRINT_STATS
-- to print the statistics of interest to the user. It then calls
-- RESET to "clean" the data space and terminates.
-- If it receives an exception from any subprogram it prints out a short
-- message and exits.
--
-- This is the main program, and it calls CHARGET, INTGET, GENERATE_ARRIVAL,
-- CREATE_SERVER, START_SIMULATION, PRINT_STATS, and RESET.
--

procedure SIMULATE is
   IATLOW : INTEGER := -10;
   IATHIGH : INTEGER := -10;
   ASTLOW : INTEGER := 18;
   ASTHIGH : INTEGER := 22;
   CAPACITY : INTEGER := 1;
   TIME: INTEGER := -1;
   START: BOOLEAN := FALSE;
   ANSWER: CHARACTER := 'a';
   CONDITION: STOPTYPE := ATIME;
   HIGHEST_SERVER, SERVER_COUNT: INTEGER := 0;
   DIST: PROBABILITY_DISTRIBUTION;
   INPUT_FILE: TEXT_IO.FILE_TYPE;
begin

TEXT_IO.OPEN (INPUT_FILE, TEXT_IO.IN_FILE,"sim.dat");

while ANSWER /= 'T' and ANSWER /= 't' and ANSWER /= 'R' and ANSWER /= 'r' and
ANSWER /= 's' and ANSWER /= 'S' loop
      ANSWER := CHARGET (INPUT_FILE);
end loop;

if ANSWER = 'T' or ANSWER = 't' then
   CONDITION := ATIME;
elsif ANSWER = 'S' or ANSWER = 's' then
   CONDITION := STARVE;
else CONDITION := REQUESTORS;
end if;

TIME := INTGET (INPUT_FILE);
if TIME < 1 then
   PUT ("That time is invalid!"); NEW_LINE;
   PUT ("Simulation aborted."); NEW_LINE;
   raise PROGRAM_ERROR;
end if;
```

```
while ANSWER /= 'u' and ANSWER /= 'U' and ANSWER /= 'e' and ANSWER /= 'E' loop
      ANSWER := CHARGET (INPUT_FILE);
end loop;

if ANSWER = 'e' or ANSWER =  ␣  then
   DIST := EXPONENTIAL;
else DIST := UNIFORM;
end if;

ANSWER := 'a';
IATLOW := INTGET (INPUT_FILE);
if IATLOW < 1 then
   PUT ("Arrival times of less than one are not allowed."); NEW_LINE;
   PUT ("Simulation aborted."); NEW_LINE;
   raise PROGRAM_ERROR;
end if;

if DIST = UNIFORM then
   IATHIGH := INTGET (INPUT_FILE);
   if IATHIGH < 1 then
      PUT ("Arrival times of less than one are not allowed."); NEW_LINE;
      PUT ("Simulation aborted."); NEW_LINE;
      raise PROGRAM_ERROR;
   end if;
else
IATHIGH := 2 ** 15 -1;
end if;
GENERATE_ARRIVAL (IATLOW, IATHIGH, DIST);

while not START loop
CREATE_SERVER (HIGHEST_SERVER, SERVER_COUNT, INPUT_FILE);
if HIGHEST_SERVER = SERVER_COUNT then
   START := TRUE;
end if;
end loop;

TEXT_IO.CLOSE (INPUT_FILE);

START_SIMULATION (DIST, IATLOW, IATHIGH, CONDITION, TIME);
PRINT_STATS;
RESET;
exception
when TEXT_IO.NAME_ERROR =>
     PUT ("Data file not found."); NEW_LINE;
when others =>
     PUT ("Simulation aborted."); NEW_LINE;
end SIMULATE;
```

```
with MATH_LIB; use MATH_LIB;
package RANDOM is
    procedure SET_SEED(N : POSITIVE);
    function RANDOM_UNIT return FLOAT;
    --returns a float >=0.0 and <1.0
    function RANDOM_INT(N : INTEGER) return INTEGER;
    --return a random integer in the range 1..N
    function EXP (MEAN: INTEGER) return INTEGER;
    -- returns a random integer exponentially distributed on MEAN.
    function UNIFORM_FUNC(LOW, HIGH: INTEGER) return INTEGER;
    --return a random integer uniformly distributed from LOW to HIGH
end RANDOM;

with CALENDAR;
use  CALENDAR;
package body RANDOM is
    MODULUS            : constant := 9317;

    type    INT_16     is range -2 ** 15 .. 2 ** 15 - 1;
    type    INT_32     is range -2 ** 31 .. 2 ** 31 - 1;
    subtype SEED_RANGE is INT_16 range 0 .. (MODULUS - 1);
    SEED, DEFAULT_SEED : SEED_RANGE;

    procedure SET_SEED(N : POSITIVE) is separate;
    function RANDOM_UNIT return FLOAT is separate;
    function RANDOM_INT(N : INTEGER) return INTEGER is separate;
    function EXP (MEAN:INTEGER) return INTEGER is separate;
    function UNIFORM_FUNC (LOW, HIGH: INTEGER) return INTEGER is separate;
begin
    DEFAULT_SEED := INT_16(INT_32(SECONDS(CLOCK)) mod MODULUS);
    SEED := DEFAULT_SEED;
end RANDOM;

separate(RANDOM)
procedure SET_SEED(N : POSITIVE) is
begin
    SEED := SEED_RANGE(N);
end SET_SEED;

separate(RANDOM)
function RANDOM_UNIT return FLOAT is
    MULTIPLIER : constant := 421;
    INCREMENT  : constant := 2073;
    RESULT     : FLOAT;
begin
    SEED := (MULTIPLIER * SEED + INCREMENT) mod MODULUS;
    RESULT := FLOAT(SEED) / FLOAT(MODULUS);
    return RESULT;
exception
    when CONSTRAINT_ERROR | NUMERIC_ERROR =>
       SEED := INT_16((MULTIPLIER * INT_32(SEED) + INCREMENT) mod MODULUS);
       RESULT := FLOAT(SEED) / FLOAT(MODULUS);
       return RESULT;
end RANDOM_UNIT;
```

```
separate(RANDOM)
function RANDOM_INT(N : INTEGER) return INTEGER is
    RESULT : INTEGER range 0 .. N;
begin
    RESULT := INTEGER(FLOAT(N) * RANDOM_UNIT + 0.5);
    return RESULT;
exception
    when CONSTRAINT_ERROR | NUMERIC_ERROR =>
       return 0;
end RANDOM_INT;

separate (RANDOM)
function EXP (MEAN:INTEGER) return INTEGER is
   TEMP: INTEGER;
   FTEMP: FLOAT;
begin
FTEMP := - LN (1.0-RANDOM_UNIT);
FTEMP := FTEMP * FLOAT (MEAN);
TEMP := INTEGER (FTEMP);
return TEMP;
end EXP;

separate(RANDOM)
function UNIFORM_FUNC(LOW,HIGH:INTEGER) return INTEGER is
   INTRESULT : INTEGER;
   RESULT: INTEGER := 0;
   SPREAD: INTEGER;
begin
   SPREAD := HIGH - LOW + 1;
   if SPREAD < 1 then
      SPREAD := 1;
   end if;
   RESULT := INTEGER (RANDOM_UNIT*FLOAT(MODULUS)) mod SPREAD;
   RESULT := RESULT + LOW;
   return RESULT;
end UNIFORM_FUNC;
```

```
-- Package for creating multiple server/multiple queue simulations

-- Authors:
--          Edward Lamie
--          Jeff Cannedy
--          Brian Pendell
-- February 1993


with TEXT_IO;
package SIMPACK is
type PROBABILITY_DISTRIBUTION is (UNIFORM,EXPONENTIAL,NORMAL,DISCRETE);
-- This type tells what distribution the random number generator will
-- use. Note that NORMAL and DISCRETE remain unimplemented at this point.

type INTEGER_32 is range -2 ** 31 .. 2 ** 31 -1;
-- A 32-bit integer type.  Used only for testing to see if MAXINT
-- (2  ** 15 -1) has been exceeded.

type EVENTCLASS is (ARRIVE, DEPART, STOPSIMULATION, SAMPLE);

-- An enumerated type describing the different kinds of events
-- that may occur during the simulation. At present, they include:
--                1. The arrival or departure of a simulation.
--                2. The termination of the simulation.
--                3. The queue sizes and various other things are sampled.

type STOPTYPE is (REQUESTORS, ATIME, STARVE);
-- An enumerated type describing the three different normal exit
-- conditions for the simulation: when a certain number of requestors
-- has left the system, when a certain time is reached on the clock,
-- and starvation, which occurs when after a certain time the system
-- prevents the arrival of any more requestors.  The system goes on
-- to finish everything already on the queues and in system, then
-- terminates.


type SCHEDULENODE;
type SCHEDULELINK is access SCHEDULENODE;
type SCHEDULENODE is record
   CLASS: EVENTCLASS;       -- type of event
   ACTIVATETIME: INTEGER;    -- when event should occur
   MAX_ARRIVAL_TIME:INTEGER;
   MIN_ARRIVAL_TIME: INTEGER;
                            -- The above two fields are used only in
generating
                            -- the initial arrival event and appear to have
                            -- outlived their usefulness.

   WHERE_AMI : INTEGER;    -- Used to tell which server the current event
                           -- is occurring at.
   FROM_SERVER: BOOLEAN;   -- Tells whether the last event
                           -- was from a server or not (This only
                           -- makes sense if the last event was an arrival).
   NEXT: SCHEDULELINK;     -- pointer for list membership
end record;
-- Describes an event that occurs during the simulation. Since all events
-- are stored on the queue, it has a pointer type associated with it.

type REQUESTNODE;
type REQUESTLINK is access REQUESTNODE;
```

```
type REQUESTNODE is record
   WHERE_NEXT: INTEGER;     -- Tells what server to visit next.
   CREATIONTIME: INTEGER;     -- When requestor was created.
   NEXT: REQUESTLINK;       -- Pointer for list membership.
end record;
-- represents a requestor.



type QUEUE_TYPE is record
     Q_PTR: REQUESTLINK;         -- A pointer to the queue proper.
     MAX_QCONTENTS: INTEGER;     -- Maximum contents.
     AVERAGE_QCONTENTS: FLOAT; -- Average contents during simulation.
     TOTAL_QENTRIES: INTEGER;  -- Total entries into the queue.
     ZERO_QENTRIES: INTEGER;     -- Number of entries waiting 0 time in queue.
     AVERAGE_QTIME: FLOAT;       -- Average time in queue during simulation.
     TOTAL_QTIME: INTEGER;         -- Total time in queue.
                                  -- This value grows very quickly, and has
                                  -- always been the first to grow above
                                  -- MAXINT during testing.
     TOTAL_QCONTENTS: INTEGER;
     CURRENT_QCONTENTS: INTEGER; -- Contents of queue when simulation stops.
end record;
-- contains the data associated with a queue

type DISTRIBUTION_RECORD;
type DISTRIBUTION_LIST is access DISTRIBUTION_RECORD;
type DISTRIBUTION_RECORD is record
     PROBABILITY: INTEGER; -- Should be in the range 0..100.
     WHERE_TO_GO: INTEGER; -- Tells what server to go to.
     NEXT: DISTRIBUTION_LIST; -- Pointer to list.
end record;

-- Every server has a distribution list associated with it, each
-- node containing the number of a server that can be reached from
-- this server and the probability that it will go there.

type SERVER_TYPE;
type SERVER_LIST_TYPE is access SERVER_TYPE;
type SERVER_TYPE is record
     SERVER_ID: INTEGER; -- Unique server id.
     NUM_PATHS: INTEGER; -- Number of branches to other servers.
     DISTRIBUTION: DISTRIBUTION_LIST; -- A list, each node of which
                                      -- contains the name of one server
                                      -- that can be reached from here,
                                      -- and the probability that it will
                                      -- go there.

     ASTLOW, ASTHIGH: INTEGER;  -- Min/Max service time for this server.
     UTILIZATION: FLOAT; -- Server utilization.
     AVERAGE_TIME_PER_ENTRY: FLOAT;
     NUMBER_OF_ENTRIES, MAX_CONTENTS, CURRENT_CONTENTS, CAPACITY: INTEGER;
     AVERAGE_CONTENTS: FLOAT;
     REQUESTORS_WAITING: INTEGER; -- number of requestors that are
                                  -- waiting on the queue.  It probably
                                  -- duplicates current contents.

     LAST_DEPARTURE: BOOLEAN; -- Set to TRUE if the last departure from
                              -- this server has been scheduled, FALSE
                              -- otherwise.
```

```
      TOTALAST: INTEGER; -- Sum of all the service times for every entry
                         -- in the server. Divide by number of entries
                         -- to obtain the average time per entry.


      NEXT: SERVER_LIST_TYPE; -- Pointer to list.
      WAIT_Q : QUEUE_TYPE; -- Pointer to the queue holding the waiting
                           -- requestors.
      PROCESS_Q: REQUESTLINK; -- A queue to hold the requestors in process.
      WHAT_KIND: PROBABILITY_DISTRIBUTION; -- Tells whether this server
                                           -- has an exponential, uniform,
                                           -- discrete or normal distribution
                                           -- for its service times.
end record;
-- Describes a server.


type SYSTEM_TYPE is record
   SAMPLES: INTEGER; -- The number of samples gathered.
   SERVER_COUNT: INTEGER; -- Number of servers
   CLOCK: INTEGER;  -- Simulation clock
   STOPTIME: INTEGER; -- stop-simulation time
   HIGHEST_SERVER: INTEGER; -- Highest server number.
   IATLOW, IATHIGH: INTEGER;  -- Lowest and highest arrival times.
   REQUESTS_PROCESSED: INTEGER; -- Number of requestors that have passed
                                -- through the system.
   TOTAL_REQUESTORS: INTEGER;   -- Total number of requestors that should
                                -- be processed before the simulation
                                -- terminates.  This field only makes sense
                                -- if the STOP_CONDITION = REQUESTORS.

   ARRIVAL_GENERATED, SERVER_CREATED, SIMULATION_RUN : BOOLEAN;
                        -- Checks to see whether a simulation can be run
                        -- safely. A simulation cannot run if an
                        -- initial arrival event has not been put on the
                        -- event queue.  Thus, ARRIVAL_GENERATED must be
                        -- TRUE.  Similarly, we must have at least one
                        -- server in the system.  Thus, SERVER_CREATED must
                        -- be TRUE.  Finally, the program cannot be
                        -- run if the data structures are still 'dirty' from
                        -- a previous run.  Thus, SIMULATION_RUN must be
                        -- FALSE.  Also, PRINT_STATS cannot execute if
                        -- a simulation has not been run, and so PRINT_STATS
                        -- will not work unless SIMULATION_RUN is TRUE.

   DISTRIBUTION_TYPE: PROBABILITY_DISTRIBUTION;
                        -- Tells what random number function to use
                        -- for the arrival times.
   CONDITION: STOPTYPE; -- Describes what condition will terminate the
                        -- simulation under normal circumstances.

   ITEM, HEAD, LAST_EVENT: SCHEDULELINK; -- Pointers to the list
                                         -- of events. ITEM is more or
                                         -- less a dummy variable while
                                         -- HEAD points to the beginning
                                         -- of the list and LAST_EVENT
                                         -- points to an event just after
                                         -- it is removed from the head of
                                         -- the list.  It will be used for
                                         -- determining which server the last
                                         -- event took place at.
```

```
end record;

-- contains all data used by the simulation system
-- but not associated with either requestors or servers.
-------------------------------------------------------------------------
-- Note: All the comments next to each of these subprograms is merely a blurb.
-- For a more complete description, see the corresponding body of the given
-- subprogram in the package body.

function CHARGET (FILE:TEXT_IO.FILE_TYPE) return CHARACTER;
        -- Gets a character from a file and returns it.
function INTGET (FILE: TEXT_IO.FILE_TYPE) return INTEGER;
        -- Same as CHARGET, but with an integer instead of a character.
function GENERATE (LOW, HIGH: INTEGER; WHAT_KIND: PROBABILITY_DISTRIBUTION)
        return INTEGER;
        -- This function accepts LOW and HIGH, which describe a boundary
        -- for a random number generator, and WHAT_KIND, which tell
        -- it what distribution, and therefore which random number
        -- generator, to use. It returns a random integer in the range
        -- LOW .. HIGH. Note that for EXPONENTIAL distributions LOW
        -- is treated as the mean and HIGH is ignored.

function SET_DESTINATION (REQ: REQUESTLINK; SERVER: SERVER_LIST_TYPE)
        return INTEGER;
        -- After a requestor (REQ) has arrived at a server (SERVER),
        -- this function is invoked to determine which server the
        -- given requestor should visit next. The function returns
        -- the ID number of the server to visit.

procedure DIST_INSERT (ITEM, LIST: in out DISTRIBUTION_LIST);
         -- Inserts an ITEM onto LIST.

procedure SERV_INSERT (NEWITEM, LISTPTR: in out SERVER_LIST_TYPE;
                       OK: in out BOOLEAN);
         -- Inserts a new server onto the list of servers, NEWITEM
         -- represents the new server and LISTPTR represents the list
         -- being inserted onto. OK is set to TRUE if the operation
         -- is successful and FALSE otherwise.

function SERV_FINDITH (LISTPTR: SERVER_LIST_TYPE; I: INTEGER)
         return SERVER_LIST_TYPE;
         -- This function accepts a pointer to a list of servers and
         -- an integer I.  It returns a pointer to the ITH server.

procedure EVENT_INSERT (ITEM, HEAD: in out SCHEDULELINK);
         -- This procedure inserts an event ITEM onto the list
         -- of events accessed by HEAD.

procedure INQUEUE (ITEM, HEAD: in out REQUESTLINK);
         -- Puts a requestor ITEM onto the list of requestors
         -- accessed by HEAD.


function QCONTENTS (HEAD: REQUESTLINK) return INTEGER;
         -- Counts the number of entries in the queue
         -- accessed by head.

procedure SCHEDULE_DEPARTURE;
         -- Schedules a departure event on the arrival queue.  It
         -- also moves a requestor off the WAIT_Q of the current server
         -- and onto the PROCESS_Q.
```

```
procedure SCHEDULE_ARRIVAL (INDEX: in INTEGER; ARRIVAL: in INTEGER;
                            FROM_SERV: BOOLEAN);
          -- An incredibly trivial procedure. It schedules an arrival
          -- at server INDEX, to occur after ARRIVAL time units have
          -- elapsed after the current time. If the requestor is departing
          -- from a server, FROM_SERV is TRUE, else FALSE.

procedure HANDLE_ARRIVAL;
          -- Handles an arrival when it occurs. Many side effects involving
          -- setting the current server, monkeying with its queues, updating
          -- statistics, etc.

procedure HANDLE_DEPARTURE;
          -- Same as HANDLE_ARRIVAL, except that it handles departures.

procedure GENERATE_ARRIVAL (LOW, HIGH : in INTEGER;
                            DISTRIBUTION: in PROBABILITY_DISTRIBUTION);
          -- SIDE EFFECT: INSERTS ARRIVAL EVENT ON SCHEDULE QUEUE.
          -- This procedure generates the first arrival event, sets
          -- the minimum and maximum arrival times for requestors, and
          -- also specifies the distribution for the arrival times.

procedure CREATE_SERVER (HIGHEST_SERVER, SERVER_COUNT: out INTEGER;
                         INPUT_FILE: in TEXT_IO.FILE_TYPE);

          -- Creates a server and puts it on the SERVER_LIST. It has
          -- to get some information to do this from the input file.

procedure PRINT_STATS;
          -- Prints out the statistics.

procedure START_SIMULATION (DISTRIBUTION: in PROBABILITY_DISTRIBUTION;
                            IATLOW, IATHIGH: in INTEGER;
                            STOP_CONDITION: in STOPTYPE;
                            STOPPER: in INTEGER);
          -- Runs the simulation itself.
procedure RESET;
          -- Initializes all data to a "clean" state.
end SIMPACK;
-------------------------------------------------
 with ADA_IO; use ADA_IO;
 with RANDOM; use RANDOM;
-------------------------------------------------
package body SIMPACK is
SERV_LIST, CURRENT_SERVER : SERVER_LIST_TYPE;
          -- A list of servers, and a pointer to the current server.
SYS : SYSTEM_TYPE;
          -- A global variable containing information about the
          -- system in general.

ERROR, FINISHED: BOOLEAN := FALSE;
          -- ERROR is used by START_SIMULATION to determine if
          -- a certain error message has been printed out. FINISHED
          -- is set to TRUE when the simulation is finished, and is
          -- FALSE the rest of the time.

-------------------------------------------------
--
--  This function, called by SIMULATE and CREATE_SERVER, pulls a character out
--  of the file described by FILE and returns it.
```

```
--  Note that it expects to read a letter a .. z or A .. Z.  If some
--  other visible character is read (such as a number or punctuation mark)
--  the function will not return anything but instead propagate the
--  PROGRAM_ERROR exception to the calling unit.  PROGRAM_ERROR will also
--  be propagated to the calling unit if the end of the file is reached before
--  a proper character is read.
--
function CHARGET (FILE: TEXT_IO.FILE_TYPE) return CHARACTER is
INPUT: CHARACTER := ASCII.BEL;
END_FILE, BAD_DATA: EXCEPTION;
begin
while CHARACTER' POS (INPUT) < 33  loop
      if TEXT_IO.END_OF_FILE (FILE) then
         raise END_FILE;
      else
         TEXT_IO.GET (FILE, INPUT);
      end if;
end loop;
if (CHARACTER' POS (INPUT) > 64 and CHARACTER' POS (INPUT) < 91)
   or (CHARACTER' POS (INPUT) > 96 and CHARACTER' POS (INPUT) < 123)
then
   return INPUT;
else raise BAD_DATA;
end if;
exception
when END_FILE =>
     PUT ("I tried to read a character but could not find it before");
     NEW_LINE;
     PUT ("I reached the end of the file."); NEW_LINE;
     raise PROGRAM_ERROR;
when BAD_DATA =>
     PUT ("An invalid character input has been entered."); NEW_LINE;
     PUT ("ASCII value of character is"); PUT (CHARACTER'POS (INPUT));
NEW_LINE;
     raise PROGRAM_ERROR;
end CHARGET;
-------------------------------------------------
--
-- This function, called by SIMULATE and CREATE_SERVER, gets an array of
-- characters from the text file FILE, converts it into an integer number
-- and returns it. Example: the string "234" is returned as
-- the integer number 234.
-- This function expects to read a number of some kind, and if a visible (
-- i.e. non-whitespace) character that is not in 0 ..9 is read, the exception
-- PROGRAM_ERROR will be propagated to the calling unit.  This will also
happen
-- if the end of file is reached before a number is read.
-- This simulation program is designed to use 16-bit arithmetic on a 32-bit
-- machine.  Thus, a number larger than 32767 will cause NUMERIC_ERROR to be
-- propagated to the calling environment.
--

function INTGET (FILE: TEXT_IO.FILE_TYPE) return INTEGER is
CHAR_ARRAY: array (1..80) of CHARACTER;
MULTIPLIER: INTEGER := 1;
COUNTER: INTEGER := 1;
STOP :BOOLEAN := FALSE;
DUMMY: CHARACTER := ASCII.BEL;
TEMP: INTEGER;
RETURN_VALUE: INTEGER := 0;
BIG1, BIG2: INTEGER_32;
```

```
END_FILE, CHARACTER_INPUT: EXCEPTION;
begin

while (not (TEXT_IO.END_OF_FILE(FILE))) and
      (not (CHARACTER' POS (DUMMY) > 47 and CHARACTER'POS (DUMMY) < 58))
      loop
      TEXT_IO.GET (FILE, DUMMY);
      if (CHARACTER' POS (DUMMY) > 32 and CHARACTER' POS (DUMMY) < 48 )
      or CHARACTER' POS (DUMMY) > 57 then
         raise CHARACTER_INPUT;
      end if;

end loop;

if (not (CHARACTER' POS (DUMMY) > 47 and CHARACTER' POS (DUMMY) < 58))
   and TEXT_IO.END_OF_FILE (FILE) then
   raise END_FILE;
end if;

CHAR_ARRAY (COUNTER) := DUMMY;
COUNTER := COUNTER + 1;
while not STOP loop
         if TEXT_IO.END_OF_LINE (FILE) or TEXT_IO.END_OF_FILE (FILE) then
            STOP := TRUE;
         else
            TEXT_IO.GET (FILE, DUMMY);

            if TEXT_IO.END_OF_LINE (FILE) or TEXT_IO.END_OF_FILE (FILE) then
               STOP := TRUE;
            end if;
            if CHARACTER'POS (DUMMY) > 47 and CHARACTER'POS (DUMMY) < 58 then
               CHAR_ARRAY(COUNTER) := DUMMY;
               COUNTER := COUNTER + 1;
               else STOP := TRUE;
               end if;
         end if; -- if we can read a character.
end loop;
COUNTER := COUNTER - 1;

while COUNTER >= 1 loop
      TEMP := (CHARACTER' POS (CHAR_ARRAY (COUNTER))) - 48;
      TEMP := TEMP * MULTIPLIER;
      BIG1 := INTEGER_32 (RETURN_VALUE);
      BIG2 := INTEGER_32 (TEMP);
      if BIG1+BIG2 > 2 ** 15 -1 then
         raise NUMERIC_ERROR;
      end if;
      RETURN_VALUE := RETURN_VALUE + TEMP;
      if MULTIPLIER < 10000 then
         MULTIPLIER := MULTIPLIER * 10;
      elsif COUNTER > 1 then
            raise NUMERIC_ERROR;
      end if;
      COUNTER := COUNTER - 1;
end loop;

return RETURN_VALUE;
exception
when CHARACTER_INPUT =>
     PUT ("I read a character where I expected a number."); NEW_LINE;
     PUT ("There must be something wrong with the input file."); NEW_LINE;
```

```
     raise PROGRAM_ERROR;
when END_FILE =>
     PUT ("I have reached the end of the input file and I could not");
     NEW_LINE;
     PUT ("Find an integer value I could accept."); NEW_LINE;
     raise PROGRAM_ERROR;
when NUMERIC_ERROR =>
     PUT ("This simulation package only allows user values of 16 bits.");
     NEW_LINE;
     PUT ("In other words, the maximum value that can be given is 32767.");
     NEW_LINE;
     raise NUMERIC_ERROR;
end INTGET;
-----------------------------------------------------
--
-- This function, called by GENERATE_ARRRIVAL, HANDLE_ARRIVAL, and
-- SCHEDULE_DEPARTURE, generates a random variable with WHAT_KIND of
-- distribution.
-- in the range LOW .. HIGH.  If WHAT_KIND is EXPONENTIAL the parameter
-- LOW is used as the mean and HIGH is ignored. If an exception of some kind
-- is raised the function will return zero to the calling unit. Otherwise
-- it will return the random value.
--

function GENERATE (LOW, HIGH: INTEGER; WHAT_KIND: PROBABILITY_DISTRIBUTION)
                   return INTEGER is
TEMP: INTEGER;
begin
    if WHAT_KIND = UNIFORM then
       TEMP :=  UNIFORM_FUNC (LOW, HIGH);
    elsif WHAT_KIND = EXPONENTIAL then
       TEMP := EXP (LOW);
    elsif WHAT_KIND = NORMAL then
       TEMP := UNIFORM_FUNC (LOW, HIGH);
       --TEMP := NORM (LOW, HIGH);
    else
       TEMP := UNIFORM_FUNC (LOW, HIGH);
       --TEMP := DISCRETE_FUNC (LOW, HIGH);
    end if;
return TEMP;
exception
when NUMERIC_ERROR =>
     PUT ("A numeric error has occurred in generate."); NEW_LINE;
     return 0;
when CONSTRAINT_ERROR =>
     PUT ("A constraint error has occurred in generate. "); NEW_LINE;
     return 0;
when PROGRAM_ERROR =>
     PUT ("A program error has occurred in generate. "); NEW_LINE;
     return 0;
when others =>
     PUT ("The generate function has failed. "); NEW_LINE;
     PUT ("Zero will be returned."); NEW_LINE;
     return 0;
end GENERATE;

-----------------------------------------------------
-- This function, called by HANDLE_ARRIVAL, determines which server the
-- requestor REQ should visit next, based on the possible servers that
-- can be reached from SERVER and the probability distribution describing
-- how probable it is that the requestor will visit a given server.
```

```
-- This information is held in  a linked list pointed to by the current
-- server's DISTRIBUTION field.

-- The function returns the identification number of the next
-- server to be visited. If an exception is raised the value zero is returned.
-- Note that when the requestors WHERE_NEXT field := 0 it means that the
-- requestor should leave the system next.  There is no server 0.
--
-- This function calls RANDOM_INT.
--

function SET_DESTINATION (REQ: REQUESTLINK; SERVER: SERVER_LIST_TYPE)
                          return INTEGER is
TEST_LIST: DISTRIBUTION_LIST;
TEST, SUM, COUNT, ID: INTEGER := 0;
DESTINATION_FOUND: BOOLEAN := FALSE;
begin
TEST_LIST := SERVER.DISTRIBUTION;
if SERVER.NUM_PATHS = 0 then
   return 0;
else
TEST := RANDOM_INT (100);
COUNT := 1;

while not DESTINATION_FOUND loop
      SUM := SUM + TEST_LIST.PROBABILITY;
      if TEST <= SUM then
        ID := TEST_LIST.WHERE_TO_GO;
        DESTINATION_FOUND := TRUE;
      end if;
      if not DESTINATION_FOUND then
         COUNT := COUNT + 1;
         TEST_LIST := TEST_LIST.NEXT;
      end if;
      if COUNT > SERVER.NUM_PATHS then
        DESTINATION_FOUND := TRUE;
        ID := 0;
      end if;
      end loop;
return ID;
end if;
exception
when others =>
     PUT ("SET_DESTINATION has failed.  Zero will be returned. ");
     NEW_LINE;
     return 0;
end SET_DESTINATION;

-------------------------------------------------

--
-- This procedure, called by CREATE_SERVER, inserts a new element ITEM onto a
-- distribution list pointed to by LIST. This is usually the DISTRIBUTION
-- field of the current server. If the procedure fails then PROGRAM_ERROR is
-- propagated to the calling unit.  Note that the exception NULL_LIST does not
-- propagate an error message.  Rather, this exception handles the special
case
-- of inserting onto an empty list.
--

procedure DIST_INSERT (ITEM, LIST: in out DISTRIBUTION_LIST) is
```

```
TEMP: DISTRIBUTION_LIST := NULL;
NULL_LIST: EXCEPTION;

begin
TEMP := LIST;
if TEMP = null then
     LIST := ITEM;
     ITEM.NEXT := null;
else
while TEMP.NEXT /= null loop
      TEMP := TEMP.NEXT;
end loop;
TEMP.NEXT := ITEM;
ITEM.NEXT := null;
end if;
exception
when NULL_LIST =>
     LIST := ITEM;
     ITEM.NEXT := null;
when others =>
     PUT ("DIST_INSERT has failed."); NEW_LINE;
     raise PROGRAM_ERROR;
end DIST_INSERT;

-------------------------------------------------

--
-- This procedure, called by CREATE_SERVER, inserts a new server NEWITEM onto
-- the list pointed to by LISTPTR. Ok is set to TRUE by this procedure if the
-- insertion is successful, FALSE otherwise.
--

procedure SERV_INSERT (NEWITEM, LISTPTR: in out SERVER_LIST_TYPE; OK: in out
BOOLEAN) is
TEMP: SERVER_LIST_TYPE := null;
COUNT: INTEGER := 1;
NULL_LIST: EXCEPTION;

begin

if LISTPTR = null then
   raise NULL_LIST;
else

TEMP := LISTPTR;
COUNT := 2;
while TEMP.NEXT /= null loop
      TEMP := TEMP.NEXT;
      COUNT := COUNT + 1;
end loop;

TEMP.NEXT := NEWITEM;
NEWITEM.NEXT := null;
NEWITEM.SERVER_ID := COUNT;

OK := TRUE;
end if;
exception
when NULL_LIST =>
     LISTPTR := NEWITEM;
     NEWITEM.NEXT := null;
```

```
     NEWITEM.SERVER_ID := 1;
     OK := TRUE;
when others =>
     PUT ("SERV_INSERT failed."); NEW_LINE;
     OK := FALSE;
end SERV_INSERT;

-------------------------------------------------
--
-- This function returns a pointer to the ITH server on the list
-- referenced by LISTPTR. If there are fewer than I things on the list
-- a pointer to the last thing is returned.  If the list is empty
-- of some other error occurs null will be returned.
-- This function is called by START_SIMULATION, HANDLE_ARRIVAL and
-- HANDLE_DEPARTURE.
--

function SERV_FINDITH (LISTPTR:  SERVER_LIST_TYPE; I: INTEGER)
         return SERVER_LIST_TYPE is
TEMP : SERVER_LIST_TYPE;
COUNT : INTEGER := 0;
EMPTY_LIST: EXCEPTION;
SHORT_LIST: EXCEPTION;

begin
if LISTPTR = null then
   raise EMPTY_LIST;
end if;
TEMP := LISTPTR;
COUNT := 1;

while COUNT < I loop

      if TEMP.NEXT = null then
        raise SHORT_LIST;
      else
         TEMP := TEMP.NEXT;
         COUNT := COUNT + 1;
      end if;
end loop;

return TEMP;
exception
when EMPTY_LIST =>
     PUT ("There are no items on the list."); NEW_LINE;
     PUT ("Null will be returned."); NEW_LINE;
     return null;
when SHORT_LIST =>
     PUT ("There are only "); PUT (COUNT);
     PUT ("items on the list."); NEW_LINE;
     PUT ("I will return a pointer to the last item on the list.");
     NEW_LINE;
     return TEMP;
when others =>
     PUT ("SERV_FINDITH has failed. Null value will be returned."); NEW_LINE;
     return null;
end SERV_FINDITH;

-------------------------------------------------

--
```

```
-- This procedure inserts a new event ITEM onto the event queue pointed
-- to by HEAD.  The insertion orders the events on the queue so that the
-- items with the smaller arrival times are closer to the top of the queue,
-- and will therefore be serviced first.  In the event of an exception
-- PROGRAM_ERROR will be propagated to the calling unit.
-- This procedure is called by SCHEDULE_ARRIVAL, SCHEDULE_DEPARTURE,
-- and START_SIMULATION.
--

procedure EVENT_INSERT (ITEM, HEAD: in out SCHEDULELINK) is
   FOUND: BOOLEAN;
   TEMPLINK1, TEMPLINK2: SCHEDULELINK;

begin
   TEMPLINK1 := HEAD;
   if HEAD = null then                                    -- list empty
      HEAD := ITEM;
      HEAD.NEXT := null;
   else
      TEMPLINK1 := HEAD;
      TEMPLINK2 := HEAD.NEXT;
      FOUND := FALSE;
      while not FOUND loop
        if ITEM.ACTIVATETIME < HEAD.ACTIVATETIME then        -- head insert
           ITEM.NEXT := HEAD;
           HEAD := ITEM;
           FOUND := TRUE;
        elsif TEMPLINK2 = null and -- tail insert 1
              ITEM.ACTIVATETIME < TEMPLINK1.ACTIVATETIME then
           ITEM.NEXT := TEMPLINK1;
           TEMPLINK1 := ITEM;
           FOUND := TRUE;
        elsif TEMPLINK2 = null and
              ITEM.ACTIVATETIME>=TEMPLINK1.ACTIVATETIME then -- tail insert 2
           ITEM.NEXT := null;
           TEMPLINK1.NEXT := ITEM;
           FOUND := TRUE;
        elsif ITEM.ACTIVATETIME < TEMPLINK2.ACTIVATETIME then   -- middle
           ITEM.NEXT := TEMPLINK2;
           TEMPLINK1.NEXT := ITEM;
           FOUND := TRUE;
        elsif FOUND = FALSE then                               -- next node
           TEMPLINK1 := TEMPLINK2;
           TEMPLINK2 := TEMPLINK2.NEXT;
        end if;
      end loop;
   end if;
exception
when others =>
     PUT ("EVENT_INSERT has failed."); NEW_LINE;
     raise PROGRAM_ERROR;
end EVENT_INSERT;

-------------------------------------------------
--
-- This procedure inserts a new requestor ITEM onto the queue pointed
-- to by HEAD.  Note that if some kind of exception occurs PROGRAM_ERROR
-- will be propagated to the calling environment.
-- This procedure is called by HANDLE_ARRIVAL and HANDLE_DEPARTURE.
--
```

```
procedure INQUEUE (ITEM, HEAD: in out REQUESTLINK) is
   FOUND: BOOLEAN;
   TEMP: REQUESTLINK;

begin
   TEMP := HEAD;
   if HEAD = null then                                    -- list empty
      HEAD := ITEM;
      HEAD.NEXT := null;
   else
      FOUND := FALSE;
      while not FOUND loop                                -- insert at rear
        if TEMP.NEXT = null then
           ITEM.NEXT := null;
           TEMP.NEXT := ITEM;
           FOUND := TRUE;
        else
           TEMP := TEMP.NEXT;
        end if;
      end loop;
   end if;
exception
when others =>
     PUT ("INQUEUE has failed. "); NEW_LINE;
     raise PROGRAM_ERROR;
end INQUEUE;

-------------------------------------------------
--
-- This function counts the number of objects on the queue (or list)
-- pointed to by HEAD and returns that number.
-- In case of an exception zero is returned.
-- This function is called by SCHEDULE_DEPARTURE, HANDLE_ARRIVAL,
--  and HANDLE_DEPARTURE.
--
function QCONTENTS (HEAD: REQUESTLINK) return INTEGER is
   COUNT: INTEGER := 0;
   TEMP: REQUESTLINK;
begin
   COUNT := 0;
   TEMP := HEAD;
   while not (TEMP = null) loop
      COUNT := COUNT + 1;
      TEMP := TEMP.NEXT;
   end loop;
return COUNT;
exception
when others =>
     PUT ("QCONTENTS has failed. Zero will be returned."); NEW_LINE;
     return 0;
end QCONTENTS;

-------------------------------------------------
--
-- Almost the first thing SCHEDULE_DEPARTURE does is check to see
-- if it CAN schedule a departure -- if the service time + the current
-- time is greater than the time when the simulation stops ( if time is the
-- stop condition for the simulation), it should not schedule a departure.
-- If the program finds that it cannot schedule a departure, it sets a flag
-- on the current server called LAST_DEPARTURE, telling itself that the
-- server will be busy with an object until the simulation terminates.
```

```
-- On the other hand, if the event passes the above test,
-- SCHEDULE_DEPARTURE schedules a departure event for the first
-- requestor on the current server's waiting queue. Having done this,
-- it moves the requestor from the current server's waiting queue to the
-- current server's processing queue, where requestors that are being
processed
-- are stored. It then updates statistics both for the current server and for
-- the current server's queue. If an exception occurs PROGRAM_ERROR is
-- propagated to the calling unit.
--
-- The procedure also checks to see if total queue time has reached MAXINT
-- (32767) if it does, than the simulation is prematurely terminated.
-- This is the only value checked here for this occurrence because
-- this field grew much faster than the other fields during testing.
--
-- This procedure is called by HANDLE_ARRIVAL and HANDLE_DEPARTURE, and calls
-- GENERATE and EVENT_INSERT.

procedure SCHEDULE_DEPARTURE is
AST: INTEGER := 0;
BIG1, BIG2, BIG3: INTEGER_32;
begin
     AST := GENERATE (CURRENT_SERVER.ASTLOW, CURRENT_SERVER.ASTHIGH,
                      CURRENT_SERVER.WHAT_KIND);


if (SYS.CLOCK+AST < SYS.STOPTIME) or (SYS.CONDITION /= ATIME) then
    SYS.ITEM := new SCHEDULENODE;
    SYS.ITEM.CLASS := DEPART;
    SYS.ITEM.ACTIVATETIME := AST + SYS.CLOCK;
    SYS.ITEM.MAX_ARRIVAL_TIME := AST;
    SYS.ITEM.MIN_ARRIVAL_TIME := 0;
    SYS.ITEM.WHERE_AMI := SYS.LAST_EVENT.WHERE_AMI;
    SYS.ITEM.FROM_SERVER := TRUE;
    SYS.ITEM.NEXT := null;

    CURRENT_SERVER.TOTALAST := CURRENT_SERVER.TOTALAST + AST;
    EVENT_INSERT (SYS.ITEM, SYS.HEAD);

elsif not CURRENT_SERVER.LAST_DEPARTURE then
      CURRENT_SERVER.LAST_DEPARTURE := TRUE;
        CURRENT_SERVER.TOTALAST := CURRENT_SERVER.TOTALAST +
                                   (SYS.STOPTIME - SYS.CLOCK);
end if;

BIG1 := INTEGER_32 (CURRENT_SERVER.WAIT_Q.TOTAL_QTIME);
BIG2 := INTEGER_32 (SYS.CLOCK);
BIG3 := INTEGER_32 (CURRENT_SERVER.WAIT_Q.Q_PTR.CREATIONTIME);
if BIG1 + (BIG2- BIG3) < 2 ** 15 - 1 then
   CURRENT_SERVER.WAIT_Q.TOTAL_QTIME := CURRENT_SERVER.WAIT_Q.TOTAL_QTIME +
                (SYS.CLOCK - CURRENT_SERVER.WAIT_Q.Q_PTR.CREATIONTIME);
else
   CURRENT_SERVER.WAIT_Q.TOTAL_QTIME := 2 ** 15 - 1;
   FINISHED := TRUE;
   if not ERROR then
   PUT ("The total queue time has just gotten as large as MAXINT."); NEW_LINE;
   PUT ("The program will be terminated prematurely.");
   ERROR := TRUE;
   end if;
end if;
if (SYS.CLOCK - CURRENT_SERVER.WAIT_Q.Q_PTR.CREATIONTIME) = 0 then
```

```
   CURRENT_SERVER.WAIT_Q.ZERO_QENTRIES :=
          CURRENT_SERVER.WAIT_Q.ZERO_QENTRIES + 1;
end if;


CURRENT_SERVER.CURRENT_CONTENTS := QCONTENTS (CURRENT_SERVER.PROCESS_Q);
CURRENT_SERVER.NUMBER_OF_ENTRIES := CURRENT_SERVER.NUMBER_OF_ENTRIES + 1;
if CURRENT_SERVER.CURRENT_CONTENTS > CURRENT_SERVER.MAX_CONTENTS then
   CURRENT_SERVER.MAX_CONTENTS := CURRENT_SERVER.CURRENT_CONTENTS;
end if;
exception
when others =>
PUT ("SCHEDULE_DEPARTURE has failed. "); NEW_LINE;
raise PROGRAM_ERROR;
end SCHEDULE_DEPARTURE;

-------------------------------------------------
--
-- An incredibly trivial procedure. It schedules an arrival
-- at server INDEX, to occur after ARRIVAL time units have
-- elapsed after the current time. If the requestor is departing
-- from a server, FROM_SERV is TRUE, else FALSE.
-- This procedure is called by HANDLE_ARRIVAL and calls EVENT_INSERT.
--
procedure SCHEDULE_ARRIVAL (INDEX: in INTEGER; ARRIVAL: in INTEGER;
                            FROM_SERV: BOOLEAN) is
begin

      SYS.ITEM := new SCHEDULENODE'(ARRIVE,ARRIVAL+SYS.CLOCK,
                                    0,0,INDEX,FROM_SERV,null);

      EVENT_INSERT (SYS.ITEM, SYS.HEAD);
exception
when others =>
     PUT ("SCHEDULE_ARRIVAL has failed. "); NEW_LINE;
end SCHEDULE_ARRIVAL;
-------------------------------------------------
--
-- This procedure handles an arrival when it occurs. The first thing
-- it does is find out which server the arrival event is occurring at and
-- make it the current server.  Next, it checks to see whether the arrival
-- event has just occurred at server 1. If it has, and it has just been
-- generated as opposed to being directed there from another server, the
-- next arrival at server one from outside the system is scheduled.
-- Statistics are then updated.  If the current server can process
-- the requestor, SCHEDULE_DEPARTURE is called.
-- In case of an exception, PROGRAM_ERROR is propagated to the calling unit.
-- This procedure is called by START_SIMULATION and calls SERV_FINDITH,
-- QCONTENTS, SCHEDULE_DEPARTURE, INQUEUE, SET_DESTINATION, GENERATE and
-- SCHEDULE_ARRIVAL.
--

procedure HANDLE_ARRIVAL is
   REQUESTOR: REQUESTLINK := null;
   NUM_ARRIVE: INTEGER := 0;
   IAT: INTEGER := 0;
begin
   CURRENT_SERVER := SERV_FINDITH (SERV_LIST,SYS.LAST_EVENT.WHERE_AMI);
   if SYS.LAST_EVENT.WHERE_AMI = 1 and not SYS.LAST_EVENT.FROM_SERVER then
      if not (SYS.CONDITION = STARVE and SYS.CLOCK > SYS.STOPTIME) then
```

```
            IAT := GENERATE (SYS.IATLOW, SYS.IATHIGH, SYS.DISTRIBUTION_TYPE);
            SCHEDULE_ARRIVAL (1, IAT, FALSE);
         else PUT ("Aborted new scheduled departure."); NEW_LINE;
         end if;
      end if;

         REQUESTOR := new REQUESTNODE'(0,SYS.CLOCK,null);
         REQUESTOR.WHERE_NEXT := SET_DESTINATION (REQUESTOR, CURRENT_SERVER);
         CURRENT_SERVER := SERV_FINDITH (SERV_LIST, SYS.LAST_EVENT.WHERE_AMI);
         INQUEUE (REQUESTOR, CURRENT_SERVER.WAIT_Q.Q_PTR);

         CURRENT_SERVER.WAIT_Q.TOTAL_QENTRIES :=
                        CURRENT_SERVER.WAIT_Q.TOTAL_QENTRIES + 1;
         CURRENT_SERVER.WAIT_Q.CURRENT_QCONTENTS :=
                        QCONTENTS (CURRENT_SERVER.WAIT_Q.Q_PTR);

         if CURRENT_SERVER.WAIT_Q.CURRENT_QCONTENTS >
                          CURRENT_SERVER.WAIT_Q.MAX_QCONTENTS then
          CURRENT_SERVER.WAIT_Q.MAX_QCONTENTS :=
                         CURRENT_SERVER.WAIT_Q.CURRENT_QCONTENTS;
         end if;

        if (CURRENT_SERVER.CURRENT_CONTENTS < CURRENT_SERVER.CAPACITY) then
           SCHEDULE_DEPARTURE;
            REQUESTOR := new REQUESTNODE;
              REQUESTOR.WHERE_NEXT := CURRENT_SERVER.WAIT_Q.Q_PTR.WHERE_NEXT;
              REQUESTOR.CREATIONTIME := CURRENT_SERVER.WAIT_Q.Q_PTR.CREATIONTIME;
              REQUESTOR.NEXT := null;
           INQUEUE (REQUESTOR, CURRENT_SERVER.PROCESS_Q);
           CURRENT_SERVER.WAIT_Q.Q_PTR := CURRENT_SERVER.WAIT_Q.Q_PTR.NEXT;
            CURRENT_SERVER.CURRENT_CONTENTS :=
                          QCONTENTS (CURRENT_SERVER.PROCESS_Q);

            if CURRENT_SERVER.CURRENT_CONTENTS > CURRENT_SERVER.MAX_CONTENTS then
               CURRENT_SERVER.MAX_CONTENTS := CURRENT_SERVER.CURRENT_CONTENTS;
            end if;

           CURRENT_SERVER.WAIT_Q.CURRENT_QCONTENTS :=
                           QCONTENTS (CURRENT_SERVER.WAIT_Q.Q_PTR);
           -- remove from queue.

        else  -- if the server is busy.
            CURRENT_SERVER.REQUESTORS_WAITING :=
                          QCONTENTS (CURRENT_SERVER.WAIT_Q.Q_PTR);

        end if; -- if server not busy
   exception
   when others =>
        PUT ("HANDLE_ARRIVAL has failed."); NEW_LINE;
        raise PROGRAM_ERROR;
   end HANDLE_ARRIVAL;
   ------------------------------------------------
   --
   -- This procedure handles the departure of requestors
   -- from a server.  The first thing it does is to find out where
   -- the departure event is supposed to occur, and make that the current
   -- server.  Having done this, it kicks the first requestor on the "processing"
   -- queue off and updates all statistics associated with this event.  Next,
   -- if there are any requestors on the queue waiting to be processed it calls
   -- SCHEDULE_DEPARTURE to pull one off the waiting queue and start processing
   -- it.  In the event of an exception PROGRAM_ERROR will be propagated to the
```

```
-- calling unit.
-- This procedure is called by START_SIMULATION, and calls SERV_FINDITH,
-- QCONTENTS, SCHEDULE_DEPARTURE, and INQUEUE.
--

procedure HANDLE_DEPARTURE is
   REQUESTOR: REQUESTLINK := null;
   NUM_ARRIVE: INTEGER := 0;
   DUMMY: INTEGER; -- UNNECESSARY IN FINAL PRODUCT.
   HUGE1, HUGE2, HUGE3: INTEGER_32;
begin

   CURRENT_SERVER := SERV_FINDITH (SERV_LIST, SYS.LAST_EVENT.WHERE_AMI);
   DUMMY := QCONTENTS (CURRENT_SERVER.WAIT_Q.Q_PTR);

   if CURRENT_SERVER.PROCESS_Q.WHERE_NEXT /= 0 then
     SCHEDULE_ARRIVAL (CURRENT_SERVER.PROCESS_Q.WHERE_NEXT,0, TRUE);
   else
      SYS.REQUESTS_PROCESSED := SYS.REQUESTS_PROCESSED + 1;
   end if;



  CURRENT_SERVER.PROCESS_Q := CURRENT_SERVER.PROCESS_Q.NEXT;
  CURRENT_SERVER.CURRENT_CONTENTS := QCONTENTS (CURRENT_SERVER.PROCESS_Q);

-- This probably won't happen here, because we are removing, and a shrinkage
-- will not exceed the maximum. RECOMMEND REMOVAL.

  if CURRENT_SERVER.CURRENT_CONTENTS > CURRENT_SERVER.MAX_CONTENTS then
     CURRENT_SERVER.MAX_CONTENTS := CURRENT_SERVER.CURRENT_CONTENTS;
  end if;

-- END REMOVAL.

if CURRENT_SERVER.REQUESTORS_WAITING > 0 then
        SCHEDULE_DEPARTURE;
         REQUESTOR := new REQUESTNODE;
           REQUESTOR.WHERE_NEXT := CURRENT_SERVER.WAIT_Q.Q_PTR.WHERE_NEXT;
           REQUESTOR.CREATIONTIME := CURRENT_SERVER.WAIT_Q.Q_PTR.CREATIONTIME;
           REQUESTOR.NEXT := null;
        INQUEUE (REQUESTOR, CURRENT_SERVER.PROCESS_Q);
         CURRENT_SERVER.CURRENT_CONTENTS :=
                QCONTENTS (CURRENT_SERVER.PROCESS_Q);
         if CURRENT_SERVER.CURRENT_CONTENTS > CURRENT_SERVER.MAX_CONTENTS then
            CURRENT_SERVER.MAX_CONTENTS := CURRENT_SERVER.CURRENT_CONTENTS;
         end if;
        CURRENT_SERVER.WAIT_Q.Q_PTR := CURRENT_SERVER.WAIT_Q.Q_PTR.NEXT;
        CURRENT_SERVER.WAIT_Q.CURRENT_QCONTENTS :=
                        QCONTENTS (CURRENT_SERVER.WAIT_Q.Q_PTR);
        -- remove from queue.

        CURRENT_SERVER.REQUESTORS_WAITING :=
                        QCONTENTS (CURRENT_SERVER.WAIT_Q.Q_PTR);
end if;

exception
when others =>
     PUT ("HANDLE_DEPARTURE has failed. "); NEW_LINE;
     raise PROGRAM_ERROR;
end HANDLE_DEPARTURE;
```

```
-------------------------------------------------
--
-- This procedure puts the first arrival event on the queue, and
-- sets the random number generator for the system, telling it how
-- often it should generate requestors. LOW and HIGH specify a range
-- of random integers and DISTRIBUTION how the numbers should be
-- distributed. Of course, if the distribution is EXPONENTIAL, HIGH has
-- no meaning and LOW refers to the mean value.  In event of an exception,
-- PROGRAM_ERROR is propagated to the calling unit.
-- This procedure is called by SIMULATE and calls GENERATE.
--

procedure GENERATE_ARRIVAL (LOW, HIGH : in INTEGER;
        DISTRIBUTION: in PROBABILITY_DISTRIBUTION) is
START: INTEGER := 0;
LOCAL_LOW,LOCAL_HIGH:INTEGER; -- LOCAL low and high values. RENAME!
begin
if LOW < 0 or HIGH <= 0 then
raise CONSTRAINT_ERROR;
end if;
if LOW > HIGH then
   PUT ("Minimum arrival time is greater than maximum arrival time.");
   NEW_LINE;
   PUT ("We will use the lower of the two values as the minimum"); NEW_LINE;
   PUT ("arrival time and the higher of the two values as the maximum.");
   NEW_LINE;
   LOCAL_LOW := HIGH;
   LOCAL_HIGH := LOW;
else
   LOCAL_LOW := LOW;
   LOCAL_HIGH := HIGH;
end if;

if DISTRIBUTION = EXPONENTIAL then
START := GENERATE (LOCAL_LOW, LOCAL_HIGH, EXPONENTIAL);
else
START := GENERATE (0, LOCAL_HIGH, UNIFORM);
end if;

SYS.ITEM := new SCHEDULENODE'(ARRIVE,START,LOCAL_HIGH,LOCAL_LOW,1,FALSE,null);
EVENT_INSERT (SYS.ITEM, SYS.HEAD);
      -- start simulation with initial arrival event.'
SYS.ARRIVAL_GENERATED := TRUE;

exception
when CONSTRAINT_ERROR =>
     PUT ("Minimum arrival time must be greater than or equal to"); NEW_LINE;
     PUT ("zero and maximum arrival time must be greater than zero.");
     NEW_LINE;
     PUT ("GENERATE_ARRIVAL procedure failed."); NEW_LINE;
     raise PROGRAM_ERROR;
when others =>
     PUT ("GENERATE_ARRIVAL procedure has failed.");
     raise PROGRAM_ERROR;
end GENERATE_ARRIVAL;

-------------------------------------------------
--
-- This procedure creates a server, specifying the minimum service time,
-- the maximum service time, and the maximum capacity of the server.
-- Having done this, it inserts the server onto SERV_LIST, which is
```

```
-- a global variable and a SIDE EFFECT.  Note that this server must read
-- INPUT_FILE to get much of its data. This file (named SIM.DAT) must be in
-- the current working directory.
-- In event of an exception, PROGRAM_ERROR is propagated to the calling
-- environment.
-- CREATE_SERVER will also check various data to see if it is accurate or not.
-- CREATE_SERVER will fail under the following conditions:
-- 1. If capacity is less than one for any server.
-- 2. If the character specifying the distribution is not 'u', 'U', 'e'
--    or 'e'.
-- 3. If any of the associated service times (minimum, maximum, or in
--    the case of EXPONENTIAL distribution mean, which is stored in the
--    minimum field ASTLOW) is less than 1.
-- 4. If the sum of the probabilities is greater than 100.
--
-- Note that if the minimum value is greater than the maximum value, they
-- will be flipped.
--
-- This procedure is called by SIMULATE and calls CHARGET, INTGET, DIST_INSERT
-- and SERV_INSERT.
--

procedure CREATE_SERVER (HIGHEST_SERVER, SERVER_COUNT: out INTEGER;
                         INPUT_FILE: in TEXT_IO.FILE_TYPE) is
LOW, HIGH, CAPACITY: INTEGER := -9000;
COUNTER: INTEGER;
NEWITEM: SERVER_LIST_TYPE;
TEMP: DISTRIBUTION_LIST;
WHAT_KIND: PROBABILITY_DISTRIBUTION;
ANSWER: CHARACTER := 'a';
LOCAL_ASTLOW, LOCAL_ASTHIGH: INTEGER; -- Local ASTLOW and ASTHIGH.
TOTAL_PROBABILITY: INTEGER := 0;
begin
if SYS.HIGHEST_SERVER = 0 then
   SYS.HIGHEST_SERVER := 1;
end if;
while ANSWER /= 'u' and ANSWER /= 'U' and ANSWER /= 'e' and ANSWER /= 'E' loop
 ANSWER := CHARGET (INPUT_FILE);
end loop;
if ANSWER = 'u' or ANSWER = 'U' then
   WHAT_KIND := UNIFORM;
else WHAT_KIND := EXPONENTIAL;
end if;

if WHAT_KIND = EXPONENTIAL then
  LOW := INTGET (INPUT_FILE);
  if LOW < 1 then
     PUT ("Associated service times of less than one are not permitted.");
     NEW_LINE;
     raise PROGRAM_ERROR;
  end if;
  HIGH := 2 ** 15 - 1;

else
  LOW := INTGET (INPUT_FILE);
  if LOW < 1 then
     PUT ("Associated service times of less than one are not permitted.");
     NEW_LINE;
     raise PROGRAM_ERROR;
  end if;
  HIGH := INTGET (INPUT_FILE);
```

```
  if HIGH < 1 then
     PUT ("Associated service times of less than one are not permitted.");
     NEW_LINE;
     raise PROGRAM_ERROR;
  end if;
end if;

      CAPACITY := INTGET (INPUT_FILE);
      if CAPACITY < 1 then
         PUT ("Capacity must be greater than zero."); NEW_LINE;
         raise PROGRAM_ERROR;
      end if;
if LOW > HIGH then
   PUT ("The entered minimum service time is greater than the maximum.");
   NEW_LINE;
   PUT ("I shall use the lower of the two values as the minimum and ");
   NEW_LINE;
   PUT ("the other as the maximum service time."); NEW_LINE;
   LOCAL_ASTLOW := HIGH;
   LOCAL_ASTHIGH := LOW;
else
   LOCAL_ASTLOW := LOW;
   LOCAL_ASTHIGH := HIGH;
end if;

NEWITEM := new SERVER_TYPE;

NEWITEM.NUM_PATHS := INTGET (INPUT_FILE);
-- PUT ("Number of paths = "); PUT (NEWITEM.NUM_PATHS); NEW_LINE;
if NEWITEM.NUM_PATHS < 0 then
   PUT ("The number of paths must be at least zero."); NEW_LINE;
   raise PROGRAM_ERROR;
end if;
if NEWITEM.NUM_PATHS = 0 then
   TEMP := new DISTRIBUTION_RECORD;
   TEMP.WHERE_TO_GO := 0;
   TEMP.PROBABILITY := 100;
   DIST_INSERT (TEMP, NEWITEM.DISTRIBUTION);
else
   TOTAL_PROBABILITY := 101;
   while TOTAL_PROBABILITY > 100 loop
      TOTAL_PROBABILITY := 0;
      for COUNTER in 1 .. NEWITEM.NUM_PATHS  loop
          TEMP := new DISTRIBUTION_RECORD;
          TEMP.WHERE_TO_GO := INTGET (INPUT_FILE);
          if TEMP.WHERE_TO_GO > SYS.HIGHEST_SERVER then
             SYS.HIGHEST_SERVER := TEMP.WHERE_TO_GO;
          end if;
          TEMP.PROBABILITY := 101;

          while TEMP.PROBABILITY > 100 or TEMP.PROBABILITY < 0 loop
            TEMP.PROBABILITY := INTGET (INPUT_FILE);
--             PUT ("probability = "); PUT (TEMP.PROBABILITY); NEW_LINE;
          end loop;
        DIST_INSERT (TEMP, NEWITEM.DISTRIBUTION);
         TOTAL_PROBABILITY := TOTAL_PROBABILITY + TEMP.PROBABILITY;
        end loop;

     if TOTAL_PROBABILITY > 100 then
          PUT ("The sum of all the probabilities for a single server");
```

```
            NEW_LINE;
            PUT ("Should not exceed 100. I'm sorry, but the simulation will");
            NEW_LINE;
            PUT ("have to be aborted."); NEW_LINE;
            raise PROGRAM_ERROR;
         end if;
      end loop; -- of while loop.
   end if; -- of if NUM_PATHS = 0.

   NEWITEM.WHAT_KIND := WHAT_KIND;
   NEWITEM.ASTLOW := LOCAL_ASTLOW;
   NEWITEM.ASTHIGH := LOCAL_ASTHIGH;
   NEWITEM.CAPACITY := CAPACITY;

   NEWITEM.NUMBER_OF_ENTRIES := 0;
   NEWITEM.UTILIZATION := 0.0;
   NEWITEM.MAX_CONTENTS := 0;
   NEWITEM.CURRENT_CONTENTS := 0;
   NEWITEM.AVERAGE_CONTENTS := 0.0;
   NEWITEM.REQUESTORS_WAITING := 0;
   NEWITEM.AVERAGE_TIME_PER_ENTRY := 0.0;
   NEWITEM.LAST_DEPARTURE := FALSE;
   NEWITEM.TOTALAST := 0;

   NEWITEM.WAIT_Q.Q_PTR := null;
   NEWITEM.WAIT_Q.TOTAL_QENTRIES := 0;
   NEWITEM.WAIT_Q.MAX_QCONTENTS := 0;
   NEWITEM.WAIT_Q.CURRENT_QCONTENTS := 0;
   NEWITEM.WAIT_Q.AVERAGE_QCONTENTS := 0.0;
   NEWITEM.WAIT_Q.ZERO_QENTRIES := 0;
   NEWITEM.WAIT_Q.AVERAGE_QTIME := 0.0;
   NEWITEM.WAIT_Q.TOTAL_QTIME := 0;
   NEWITEM.WAIT_Q.TOTAL_QCONTENTS := 0;
   SERV_INSERT (NEWITEM, SERV_LIST, SYS.SERVER_CREATED); -- POSSIBLE BAD MOVE
                                                         -- using SYS.SERVER_CREATED.


   if SYS.SERVER_CREATED then
      CURRENT_SERVER := NEWITEM;
   end if;

   SYS.SERVER_COUNT := SYS.SERVER_COUNT + 1;
   SERVER_COUNT := SYS.SERVER_COUNT;  -- parameter assignments.
   HIGHEST_SERVER := SYS.HIGHEST_SERVER;
   exception
   when others =>
        PUT ("CREATE_SERVER procedure has failed."); NEW_LINE;
        raise PROGRAM_ERROR;
   end CREATE_SERVER;

   ------------------------------------------------------
   -- This procedure prints out the statistics of interest to a user.
   -- it may only be used after a simulation is run.
   --
   -- This procedure is called by SIMULATE.
   procedure PRINT_STATS is
   NO_SIMULATION : EXCEPTION;
   begin
   if not SYS.SIMULATION_RUN then
      raise NO_SIMULATION;
   end if;
```

```
PUT ("Simulation Stopped at Time:  "); PUT (SYS.CLOCK);
NEW_LINE;

CURRENT_SERVER := SERV_LIST;
PUT (SYS.SERVER_COUNT); PUT (" servers in this run."); NEW_LINE;
PUT ("SERVER   CAPACITY  ENTRIES  AVG. TIME   MAX    CURRENT   AVG.
UTIL. ");
NEW_LINE;
PUT ("                          PER  ENTRY     \     |    /          ");
NEW_LINE;
PUT ("                                          CONTENTS            ");
NEW_LINE;
for COUNT in 1..SYS.SERVER_COUNT loop
    PUT (CURRENT_SERVER.SERVER_ID,2);
    PUT (CURRENT_SERVER.CAPACITY,12);
    PUT (CURRENT_SERVER.NUMBER_OF_ENTRIES,9);
    PUT (CURRENT_SERVER.AVERAGE_TIME_PER_ENTRY,10);
    PUT (CURRENT_SERVER.MAX_CONTENTS,8);
    PUT (CURRENT_SERVER.CURRENT_CONTENTS,7);
    PUT (CURRENT_SERVER.AVERAGE_CONTENTS,9);
    PUT (CURRENT_SERVER.UTILIZATION*100.0,9); PUT (" %"); NEW_LINE;
    CURRENT_SERVER := CURRENT_SERVER.NEXT;
end loop;

CURRENT_SERVER := SERV_LIST;
NEW_LINE; NEW_LINE; NEW_LINE;

PUT ("QUEUE   ENTRIES   0-ENTRIES   AVG. TIME      MAX.   CURRENT     AVG.");
NEW_LINE;
PUT ("                                                CONTENTS        ");
NEW_LINE;
for COUNT in 1..SYS.SERVER_COUNT loop
PUT (CURRENT_SERVER.SERVER_ID,2);
PUT (CURRENT_SERVER.WAIT_Q.TOTAL_QENTRIES,12);
PUT (CURRENT_SERVER.WAIT_Q.ZERO_QENTRIES,12);
PUT (CURRENT_SERVER.WAIT_Q.AVERAGE_QTIME,9);
PUT (CURRENT_SERVER.WAIT_Q.MAX_QCONTENTS,10);
PUT (CURRENT_SERVER.WAIT_Q.CURRENT_QCONTENTS,9);
PUT (CURRENT_SERVER.WAIT_Q.AVERAGE_QCONTENTS,9);
NEW_LINE;
CURRENT_SERVER := CURRENT_SERVER.NEXT;
end loop;
NEW_LINE; NEW_LINE; NEW_LINE; NEW_LINE;

exception
when NO_SIMULATION =>
     PUT ("You must run a simulation before I can print out any statistics.");
     NEW_LINE;
     PUT ("PRINT_STATS procedure failed."); NEW_LINE;
when others =>
     PUT ("PRINT_STATS procedure failed."); NEW_LINE;
end PRINT_STATS;
-------------------------------------------------
-- This procedure starts the simulation, generating the termination event
-- and the first sample event.  After this it enters a loop and processes
-- every event as it comes off the event queue until FINISHED := TRUE.
-- To do this it will have to call HANDLE_ARRIVAL and HANDLE_DEPARTURE
-- to handle the ARRIVAL and DEPARTURE events. After FINISHED is TRUE
-- the procedure calculates all the means from data gathered during the
-- simulation, and also factors in the waiting times of the requestors still
```

```
-- on the queues.  Server utilization is also derived at this time.
-- There are a number of exceptions that will prevent this program from
working.
-- BAD_WORKSPACE is raised if a simulation has just been run and the data
-- structures are still "dirty" from the run.  NO_ARRIVAL is raised
-- if no initial arrival event has been generated, and NO_SERVER is raised
-- if there is no server to do processing.  In all, There are  5 conditions
-- that must occur for this procedure to work:
-- 1. The necessary data structures must exist and they must be "clean."
-- 2. An initial arrival event must be on the event queue.
-- 3. At least one server must be on the server list.
-- 4. Time for simulation must be greater than zero, if the ending condition
--    is either STARVE or TIME. If it isn't the number of requestors that must
--    leave the system before termination must be greater than zero.
-- 5. There must be the RIGHT number of servers. An exception will be
generated
--    if, for instance, a user creates pointers to seven servers but only
--    generates four.

-- This procedure is called by SIMULATE and calls HANDLE_ARRIVAL,
-- HANDLE_DEPARTURE, SERV_FINDITH and EVENT_INSERT.
--
procedure START_SIMULATION (DISTRIBUTION: in PROBABILITY_DISTRIBUTION;
                            IATLOW, IATHIGH: in INTEGER;
                            STOP_CONDITION: in STOPTYPE;
                            STOPPER: in INTEGER) is
TIME: INTEGER;
REQUESTOR: REQUESTLINK := null;
SINTERVAL: INTEGER := 0; -- determines the interval at which samples are
                         -- taken.
HUGE1, HUGE2, HUGE3 : INTEGER_32;
NO_SERVER, NO_ARRIVAL, BAD_WORKSPACE, NOT_ENOUGH_SERVERS: EXCEPTION;
NOT_ENOUGH_REQUESTORS: EXCEPTION;
-------------------------------------

begin

SYS.DISTRIBUTION_TYPE := DISTRIBUTION;
SYS.CONDITION := STOP_CONDITION;

if SYS.CONDITION /= REQUESTORS then
   SYS.STOPTIME := STOPPER;
else SYS.TOTAL_REQUESTORS := STOPPER;
end if;

if SYS.SIMULATION_RUN then
   raise BAD_WORKSPACE;
elsif not SYS.ARRIVAL_GENERATED then
   raise NO_ARRIVAL;
elsif not SYS.SERVER_CREATED then
   raise NO_SERVER;
elsif SYS.SERVER_COUNT < SYS.HIGHEST_SERVER then
      raise NOT_ENOUGH_SERVERS;
else
SYS.IATLOW := IATLOW;
SYS.IATHIGH := IATHIGH;

if SYS.CONDITION = ATIME then
-- Schedule simulation termination. HASTA LA VISTA!
SYS.ITEM := new SCHEDULENODE'(STOPSIMULATION,SYS.STOPTIME,0,0,0,FALSE,null);
EVENT_INSERT (SYS.ITEM, SYS.HEAD);
```

```
          -- master scheduler stop simulation
end if;

-- Schedule the first sampling event.
if SYS.CONDITION /= REQUESTORS then
   SINTERVAL := GENERATE ((SYS.STOPTIME / 20), (SYS.STOPTIME / 10), UNIFORM);
else
   SINTERVAL := GENERATE (((SYS.IATLOW * SYS.TOTAL_REQUESTORS) / 20),
                ((SYS.IATLOW * SYS.TOTAL_REQUESTORS) / 10), UNIFORM);
end if;
if SINTERVAL < 1 then
   SINTERVAL := 1;
end if;
-- PUT ("SINTERVAL = "); PUT (SINTERVAL); NEW_LINE;

SYS.ITEM := new SCHEDULENODE;
SYS.ITEM.CLASS := SAMPLE;
SYS.ITEM.ACTIVATETIME := SINTERVAL;
SYS.ITEM.MAX_ARRIVAL_TIME := 0;
SYS.ITEM.MIN_ARRIVAL_TIME := 0;
SYS.ITEM.FROM_SERVER := FALSE;
SYS.ITEM.NEXT := null;

EVENT_INSERT (SYS.ITEM, SYS.HEAD);

CURRENT_SERVER := SERV_FINDITH (SERV_LIST,1);


while not FINISHED loop
      if SYS.HEAD /= null and SYS.CLOCK < SYS.HEAD.ACTIVATETIME
      then
        SYS.CLOCK := SYS.HEAD.ACTIVATETIME;
      end if;
      if SYS.REQUESTS_PROCESSED >= SYS.TOTAL_REQUESTORS-1 and
         SYS.CONDITION = REQUESTORS then
         SYS.STOPTIME := SYS.LAST_EVENT.ACTIVATETIME; -- May be wrong.
         FINISHED := TRUE;
      elsif SYS.HEAD.CLASS = ARRIVE then
         SYS.LAST_EVENT := SYS.HEAD;
        SYS.HEAD := SYS.HEAD.NEXT;        -- remove top node
        HANDLE_ARRIVAL;
      elsif SYS.HEAD.CLASS = DEPART then
         SYS.LAST_EVENT := SYS.HEAD;
        SYS.HEAD := SYS.HEAD.NEXT;
        HANDLE_DEPARTURE;
      elsif SYS.HEAD.CLASS = STOPSIMULATION then
           SYS.STOPTIME := SYS.HEAD.ACTIVATETIME;
           SYS.HEAD := SYS.HEAD.NEXT;
           FINISHED := TRUE;
      elsif SYS.HEAD.CLASS = SAMPLE then
           SYS.HEAD := SYS.HEAD.NEXT;
            if SYS.CLOCK > SYS.STOPTIME and SYS.CONDITION = STARVE then
               FINISHED := TRUE;
               SYS.ITEM := SYS.HEAD;
               -- This next loop determines whether the system has starved
               -- or not.  What we do is check every item on the event queue
               -- and if there are no arrivals or departures on the queue
               -- then the system has starved and the simulation should
               -- terminate.
               while SYS.ITEM /= null loop
```

```
                    if SYS.ITEM.CLASS = ARRIVE or SYS.ITEM.CLASS = DEPART
then
                       FINISHED := FALSE;
                    end if;
                    SYS.ITEM := SYS.ITEM.NEXT;
                end loop;
             end if;
             if not FINISHED then
             for COUNTER in 1..SYS.SERVER_COUNT loop
             CURRENT_SERVER := SERV_FINDITH (SERV_LIST, COUNTER);
            CURRENT_SERVER.WAIT_Q.TOTAL_QCONTENTS :=
                        CURRENT_SERVER.WAIT_Q.TOTAL_QCONTENTS +
                        CURRENT_SERVER.WAIT_Q.CURRENT_QCONTENTS;
             end loop;

            SYS.SAMPLES := SYS.SAMPLES + 1;

            SYS.ITEM := new SCHEDULENODE'
                        (SAMPLE,SYS.CLOCK+SINTERVAL,0,0,0,FALSE,null);
            EVENT_INSERT (SYS.ITEM, SYS.HEAD);

          end if; -- if we are not finished in sample.
      end if;  -- of the outermost if.
end loop;

SYS.STOPTIME := SYS.CLOCK;
for COUNTER in 1..SYS.SERVER_COUNT loop

CURRENT_SERVER := SERV_FINDITH (SERV_LIST, COUNTER);

if SYS.SAMPLES > 0 then
CURRENT_SERVER.WAIT_Q.AVERAGE_QCONTENTS := FLOAT
                            (CURRENT_SERVER.WAIT_Q.TOTAL_QCONTENTS) /
                            FLOAT (SYS.SAMPLES);
else
CURRENT_SERVER.WAIT_Q.AVERAGE_QCONTENTS := 0.0;
end if;


REQUESTOR := CURRENT_SERVER.WAIT_Q.Q_PTR;
while REQUESTOR /= null loop
      HUGE1 := INTEGER_32 (CURRENT_SERVER.WAIT_Q.TOTAL_QTIME);
      HUGE2 := INTEGER_32 (SYS.CLOCK);
      HUGE3 := INTEGER_32 (REQUESTOR.CREATIONTIME);
      if HUGE1 + (HUGE2 - HUGE3) < 2 ** 15 - 1 then
         CURRENT_SERVER.WAIT_Q.TOTAL_QTIME :=
                            CURRENT_SERVER.WAIT_Q.TOTAL_QTIME +
                            (SYS.CLOCK - REQUESTOR.CREATIONTIME);
         REQUESTOR := REQUESTOR.NEXT;
      elsif not ERROR then
         PUT ("The time in queue has reached MAXINT."); NEW_LINE;
         CURRENT_SERVER.WAIT_Q.TOTAL_QTIME := 2 ** 15 - 1;
         REQUESTOR := null;
         ERROR := TRUE;
      else
         CURRENT_SERVER.WAIT_Q.TOTAL_QTIME := 2 ** 15 -1;
         ERROR := TRUE;
         REQUESTOR := null;
      end if;

end loop;
```

```
if CURRENT_SERVER.WAIT_Q.TOTAL_QENTRIES > 0 then
CURRENT_SERVER.WAIT_Q.AVERAGE_QTIME :=
                        FLOAT (CURRENT_SERVER.WAIT_Q.TOTAL_QTIME) /
                        FLOAT (CURRENT_SERVER.WAIT_Q.TOTAL_QENTRIES);
else
CURRENT_SERVER.WAIT_Q.AVERAGE_QTIME := 0.0;
end if;
if CURRENT_SERVER.NUMBER_OF_ENTRIES > 1 then
CURRENT_SERVER.NUMBER_OF_ENTRIES :=  CURRENT_SERVER.NUMBER_OF_ENTRIES -
                                     CURRENT_SERVER.CURRENT_CONTENTS;
CURRENT_SERVER.AVERAGE_TIME_PER_ENTRY := FLOAT (CURRENT_SERVER.TOTALAST) /
                                     FLOAT(CURRENT_SERVER.NUMBER_OF_ENTRIES);
else
CURRENT_SERVER.AVERAGE_TIME_PER_ENTRY := 0.0;
end if;
CURRENT_SERVER.UTILIZATION := CURRENT_SERVER.AVERAGE_TIME_PER_ENTRY *
                              FLOAT (CURRENT_SERVER.NUMBER_OF_ENTRIES) /
                              FLOAT (CURRENT_SERVER.CAPACITY) /
                              FLOAT (SYS.STOPTIME);

CURRENT_SERVER.AVERAGE_CONTENTS := CURRENT_SERVER.UTILIZATION *
                                   FLOAT (CURRENT_SERVER.CAPACITY);
end loop;

SYS.SIMULATION_RUN := TRUE;
end if; -- of if nothing wrong.
 exception
 when STORAGE_ERROR =>
      PUT ("Storage error exception in START_SIMULATION"); NEW_LINE;
 when PROGRAM_ERROR =>
      PUT ("Program error exception in START_SIMULATION"); NEW_LINE;
 when NO_SERVER =>
     PUT ("You must generate at least one server with CREATE_SERVER");
     NEW_LINE;
     PUT ("before you can run a simulation."); NEW_LINE;
     PUT ("START_SIMULATION procedure failed. "); NEW_LINE;
when NUMERIC_ERROR =>
     PUT ("Numeric error occurred in START_SIMULATION."); NEW_LINE;
when NO_ARRIVAL =>
     PUT ("You must generate the first arrival with GENERATE_ARRIVAL");
     NEW_LINE;
     PUT ("before you can run a simulation."); NEW_LINE;
     PUT ("START_SIMULATION procedure failed. "); NEW_LINE;
when BAD_WORKSPACE =>
     PUT ("The data structures are still 'dirty' from the last run.");
     NEW_LINE;
     PUT ("I dare not run again until they have been cleaned up. ");
     NEW_LINE;
     PUT ("START_SIMULATION procedure failed. "); NEW_LINE;
when CONSTRAINT_ERROR =>
     PUT ("Time must be greater than zero."); NEW_LINE;
     PUT ("START_SIMULATION procedure failed."); NEW_LINE;
when NOT_ENOUGH_SERVERS =>
     PUT ("You have created pointers to "); PUT (SYS.HIGHEST_SERVER);
     NEW_LINE;
     PUT ("servers but created only "); PUT (SYS.SERVER_COUNT);
     if SYS.SERVER_COUNT = 1 then
        PUT (" server.");
     else PUT (" servers.");
     end if;
```

```
     NEW_LINE;
     PUT ("START_SIMULATION procedure failed.");
when others =>
     PUT ("START_SIMULATION procedure failed. "); NEW_LINE;
end START_SIMULATION;

-----------------------
--
-- This procedure resets the simulation package so that it will run
-- again.
--
-- It is called by SIMULATE.
--
procedure RESET is
begin
FINISHED := FALSE;
ERROR := FALSE;
SYS.TOTAL_REQUESTORS := 0;
SYS.REQUESTS_PROCESSED := 0;
SYS.DISTRIBUTION_TYPE := UNIFORM;
SYS.ARRIVAL_GENERATED := FALSE;
SYS.SERVER_CREATED := FALSE;
SYS.SIMULATION_RUN := FALSE;
CURRENT_SERVER := null;
SERV_LIST := null;
SYS.HIGHEST_SERVER := 0;
SYS.SERVER_COUNT := 0;
SYS.SAMPLES := 0;
SYS.CLOCK := 0;
SYS.STOPTIME  := 0;
SYS.ITEM := null;
SYS.HEAD := null;

end RESET;

--------------------------------------

begin  -- SIMPACK initialization code.

RESET;

end SIMPACK;
```